Math. Prog. Comp. (2018) 10:303–332
https://doi.org/10.1007/s12532-017-0131-4

FULL LENGTH PAPER

Asynchronously parallel optimization solver for finding multiple minima

Jeffrey Larson[1] · **Stefan M. Wild**[1]

Received: 1 April 2016 / Accepted: 23 September 2017 / Published online: 16 February 2018

Abstract We propose and analyze an asynchronously parallel optimization algorithm for finding multiple, high-quality minima of nonlinear optimization problems. Our multistart algorithm considers all previously evaluated points when determining where to start or continue a local optimization run. Theoretical results show that when there are finitely many minima, the algorithm almost surely starts a finite number of local optimization runs and identifies every minimum. The algorithm is applicable to general optimization settings, but our numerical results focus on the case when derivatives are unavailable. In numerical tests, a PYTHON implementation of the algorithm is shown to yield good approximations of many minima (including a global minimum), and this ability is shown to scale well with additional resources. Our implementation's time to solution is shown also to scale well even when the time to perform the function evaluation is highly variable. An implementation of the algorithm is available in the libEnsemble library at https://github.com/Libensemble/libensemble.

Keywords Parallel optimization algorithms · Multistart · Global optimization · Derivative-free optimization · Concurrent function evaluations

Mathematics Subject Classification 65K05 · 90C26 · 90C30 · 90C56

✉ Stefan M. Wild
wild@anl.gov

Jeffrey Larson
jmlarson@anl.gov

[1] Argonne National Laboratory, 9700 S. Cass Ave., Bldg. 240, Lemont, IL 60439, USA

1 Introduction

This paper addresses the problem of finding high-quality minima of

$$\begin{aligned} &\underset{x \in \mathbb{R}^n}{\text{minimize}} \; f(x) \\ &\text{subject to: } x \in \mathscr{D}, \end{aligned} \tag{1}$$

when $\mathscr{D}$ is compact, concurrent evaluations of f are possible, and relatively little is known about f a priori. We consider the case where the number of possible concurrent evaluations c is small, for example, when each evaluation of f requires a significant amount of computational resources. In this paper, we propose and analyze the theoretical and practical characteristics of an algorithm for finding high-quality minima of (1), even when derivatives of f are unavailable. Although an algorithm's best-found local minimum provides an obvious approximation of the global minimizer of (1), we differentiate the task of finding multiple high-quality minima from that of approximate global optimization.

In many applications analyzing nonglobal minima has value. In physics, for example, the various minimizers of a potential energy surface correspond to different ground and excited states; in chemistry, different dimer structures can be found by examining minimizers from a geometry optimization problem [4]. In this paper, we consider *high-quality minima* to be those minima with lowest function values; however, high-quality can refer to more than just function value. For example, a problem may have local minima that need to be compared by using criteria not known at the time of optimization or criteria that are less quantifiable, such as computational architects who "are not necessarily interested in finding the single most optimal design…but rather are interested in being able to visualize and evaluate a range of well-performing solutions" [3].

In this paper, we present an asynchronously parallel multistart algorithm for finding local minima of a nonlinear optimization problem with a compact domain $\mathscr{D}$. The algorithm samples the domain $\mathscr{D}$ and starts local optimization runs from promising points. The sampling is necessary when one cannot make additional assumptions about the function f [30]. Because we view local optimization as presenting a significant sequential expense that typically cannot be mitigated by increased concurrency, the algorithm seeks to minimize the number of local optimization runs performed. Ideally, each minimum will be identified by a single local optimization run. We prove in Sect. 3 that, under certain assumptions, the algorithm almost surely identifies all minima for (1) while starting only a finite number of local optimization runs. Since many local optimization methods converge only asymptotically, we consider a minimum to be *identified* if it has been computed exactly or if a local optimization run is converging asymptotically to it.

The development of the algorithm is motivated by two observations. We first observe that most local optimization methods do not efficiently utilize concurrent function evaluations. That said, we want to use state-of-the-art (sequential) local optimization methods because many are designed to handle inexact derivative information, exploit problem structure, and are robust to noise and stochasticity. The second observation

is that many heuristics either do not scale well in their time to solution when given additional resources or perform poorly with few resources. (Note that random sampling usually performs poorly with few resources, but its performance scales perfectly with the number of concurrent function evaluations.)

We show that an implementation of the proposed algorithm, an asynchronously parallel optimization solver for finding multiple minima (APOSMM), is able to find high-quality minima for a large set of synthetic test problems. APOSMM is written in PYTHON using the NUMPY and SCIPY libraries; the message-passing interface MPI is used to coordinate the asynchronous evaluation of f by a collection of *workers*. A central *manager* tells the workers which points to evaluate and, as evaluated points are returned from workers, advances local optimization runs by using a collection of *custodians*. Note that the custodians do not perform any function evaluations but only report the next point desired by some local optimization algorithm to the manager. The custodians can find the next point requested by any local optimization method. The implementation is detailed in Sect. 4.

The theoretical results assume nothing about the availability of ∇f, and therefore the algorithm and the APOSMM implementation are also applicable to problems where derivatives are unavailable. We present extensive numerical tests in Sect. 5 for this derivative-free case and highlight the scalability of our implementation. We find that the algorithm's (and implementation's) efficient use of previous function evaluations makes it especially well suited for problems where ∇f is unavailable.

Before reviewing the literature, we briefly establish the notation that will be used to describe the asynchronous manager–worker algorithm. We follow the convention that finite sets are denoted by capital Roman letters while (possibly) infinite sets are denoted by capital calligraphic letters. Let X^* denote the set of all local minima of f within $\mathscr{D}$, and let $x^*_{(i)}$ be the minima in X^* with the ith smallest function value (with ties broken as needed). A local optimization run is *active* if it has been started but has not yet ruled an iterate to be a local minimum. Q_k will denote the queue of points from active local optimization runs that have yet to be given to a worker at iteration k, and $\mathscr{R}_S$ will represent a random stream of points uniformly drawn from $\mathscr{D}$. Let $H_k = S_k \bigcup L_k$ be the set of all points evaluated by APOSMM from iteration 1 to k, consisting of points S_k randomly sampled and points L_k generated by local optimization runs. Let A_k denote the set of points that are within active local optimization runs and have been, or are currently being, evaluated at iteration k. That is, A_k contains all points evaluated in the course of local optimization runs that have not ruled an iterate to be a local minimum by iteration k.

2 Background

Many methods exist for finding minima for general nonlinear optimization problems of the form (1). In this section, we highlight methods that utilize concurrent evaluations of the objective function and methods that explicitly seek a global minimum of (1). Methods of the former type can be restarted at different points in the domain $\mathscr{D}$. Convergent methods of the latter type must sufficiently explore the domain $\mathscr{D}$ [30].

Several derivative-free implementations of algorithms can utilize concurrent evaluations of the objective function. pVTDIRECT [14–16] is a parallel implementation of the "dividing rectangles" algorithm DIRECT [19]; pVTDIRECT is designed to efficiently evaluate the same set of points independent of c, the number of possible concurrent function evaluations. The asynchronous parallel pattern search algorithm [17] also evaluates candidate search directions concurrently. Implementations of this algorithm include APPSPACK [11] and HOPSPACK [25]. In [1], an extension of mesh-adaptive direct search (MADS) decomposes the problem (1) into a finite number of smaller-dimensional subproblems, each of which is solved by a single MADS instance. These subproblem instances are independent, and thus there is no synchronization among iterations of the concurrent instances of MADS. In a similar fashion, the authors of [8,22] describe how pattern searches around several candidate solutions can be performed concurrently.

Olsson [24] studies approaches for utilizing concurrent evaluations of the objective for model-based, local, derivative-free methods. Vanden Berghen [31] outlines parallel extensions of the CONDOR algorithm for constrained, model-based optimization.

Many heuristic approaches also exist for solving (1). Two of the most popular heuristics are the particle swarm, pattern search hybrid [32] and the covariance matrix adaptation evolution strategy CMA-ES [13]. In Sect. 5.4 we compare our work with a version of CMA-ES that allows for concurrent evaluations.

Methods capable of identifying multiple minima for problems such as (1) have been studied in the literature. For example, the performance of a collection of implementations (including pVTDIRECT) on three relatively higher-dimensional problems ($n \in \{56, 57\}$) with many minima is studied in [7], and MADS is used in [10] to identify low-melting compositions in a system containing many chemical components. GLODS [5] is a multistart implementation that is also designed to identify many minima. In Sect. 5.4 we compare an implementation of Algorithm 1 with both GLODS and pVTDIRECT.

3 Algorithmic details and theory

The asynchronously parallel algorithm we present in this section is influenced by three previous algorithms: Multi-Level Single Linkage (MLSL) [27,28], Maximum Information from Previous Evaluations (MIPE) [33], and the batch algorithm for multiple local minima (BAMLM) [21]. Although MLSL is a multistart algorithm with theoretical properties similar to those of our algorithm, it is lacking for a few reasons. MLSL does not consider points generated during previous local optimization runs when deciding where to start subsequent runs. Function values computed during past runs can provide useful information, especially when the objective function is expensive to evaluate. Also, MLSL implicitly assumes that local optimization runs must be completed before subsequent runs are started. MIPE addresses the first limitation of MLSL, but it starts runs as needed without considering whether resources are available. The BAMLM algorithm is a direct precursor to the algorithm proposed here, although BAMLM's theoretical results are based on the assumption that any local optimization run that is started will converge to a minimum in a finite number of

function evaluations. In contrast to these algorithms, APOSMM utilizes all previously evaluated points when deciding where to start subsequent local optimization runs, honors limitations on the number of concurrent evaluations that can occur, and has theoretical results based on the assumption that local optimization runs may converge only asymptotically to a minimum.

We show that under certain assumptions on the function and the local optimization method, APOSMM will almost surely start a finite number of local optimization runs and every local minimum will be identified: the minimum is either calculated explicitly or has a single local optimization run converging to it.

An event happens *almost surely* if it happens with probability one. If the set of possible events is finite, then almost surely means the event occurs. If the set of possible events is infinite, then an almost sure event may not occur, but the probability of the event not occurring is smaller than any fixed positive value [29]. Therefore, our results will show that for any random sample stream $\mathscr{R}_S$ of points drawn uniformly from the domain $\mathscr{D}$, APOSMM will start finitely many runs and identify every minimum with probability one.

3.1 Algorithmic description

Our parallel multistart algorithm is designed to find many local minima of the nonlinear optimization problem (1). The compact domain $\mathscr{D}$ is explored by evaluating points randomly sampled from it, and then local optimization runs are started at promising points. As function values are asynchronously returned from workers evaluating the objective function, additional local optimization runs may be started, and active runs may be merged. Points, either randomly sampled from $\mathscr{D}$ or from local optimization runs, are then given to workers to evaluate.

The MLSL algorithm [27,28] decides where to start local optimization runs in a similar fashion, but it considers only points it has randomly sampled by iteration k (S_k) when deciding where to start local optimization runs on iteration k. Our algorithm uses all previously evaluated points by considering points in S_k and all points evaluated by past local optimization runs, L_k. Points in L_k local optimization runs that are active (i.e., those runs that have not ruled an iterate to be a local minimum) at iteration k are included in the set of active local optimization points $A_k \subseteq L_k$.

Table 1 lists the conditions that a previously evaluated point $\hat{x}$ must satisfy before it is used as a starting point for a local optimization run. In general, runs are started from points $\hat{x}$ that are not within a distance r_k of points with lower function values, have not already started a run, and are not too close to the domain boundary. We define the threshold $r_k > 0$ on iteration k to be

$$r_k = \frac{1}{\sqrt{\pi}} \sqrt[n]{\Gamma\left(1 + \frac{n}{2}\right) \text{vol}\left(\mathscr{D}\right) \frac{5 \log |S_k|}{S_k}}. \tag{2}$$

This is similar to the thresholds used by MLSL, MIPE, and BAMLM.

Table 1 Logical conditions to determine when to start a local optimization run after k evaluations

(L1)	$\hat{x} \in L_k$ and $\nexists x \in L_k$ with $\left[\left\|\hat{x} - x\right\| \leq r_k \text{ and } f(x) < f(\hat{x})\right]$
(S1)	$\hat{x} \in S_k$ and $\nexists x \in L_k$ with $\left[\left\|\hat{x} - x\right\| \leq r_k \text{ and } f(x) < f(\hat{x})\right]$
(L2)	$\hat{x} \in L_k$ and $\nexists x \in S_k$ with $\left[\left\|\hat{x} - x\right\| \leq r_k \text{ and } f(x) < f(\hat{x})\right]$
(S2)	$\hat{x} \in S_k$ and $\nexists x \in S_k$ with $\left[\left\|\hat{x} - x\right\| \leq r_k \text{ and } f(x) < f(\hat{x})\right]$
(L3)	$\hat{x} \in L_k$ has not started a local optimization run
(S3)	$\hat{x} \in S_k$ has not started a local optimization run
(L4)	$\hat{x} \in L_k$ is at least a distance μ from the domain boundary
(S4)	$\hat{x} \in S_k$ is at least a distance μ from the domain boundary
(L5)	$\hat{x} \in L_k$ is at least a distance ν from any known local min
(S5)	$\hat{x} \in S_k$ is at least a distance ν from any known local min
(L6)	$\hat{x} \in L_k$ is not in A_k
(L7)	$\hat{x} \in L_k$ has not been ruled stationary
(L8)	$\hat{x} \in L_k$ is on an r_k-descent path in H_k for some $x \in S_k$ satisfying (S2)–(S5)

Note that Line 13 of Algorithm 1 requires knowledge of each local optimization run's candidate minima (i.e., the local optimization method's estimate of the local minimum) for each run. We consider a run's starting point as the first candidate minimum, even if that point came from S_k.

Algorithm 1: Asynchronously parallel algorithm for finding multiple minima.

input: c workers; a local optimization method; tolerances $\mu, \nu > 0$; a uniform random stream $\mathscr{R}_S \subset \mathscr{D}$.

1 Initialize $Q_k = X_0^* = H_0 = L_0 = S_0 = A_0 = \emptyset$ and $k = 0$.
2 **for** $w = \{1, \ldots, c\}$ **do**
3 Give w a point from $\mathscr{R}_S$ at which to evaluate f.
4 **while true do**
5 (Possibly wait to) Receive from worker w that has evaluated its point $\tilde{x}$ (taking from the longest-waiting worker if multiple are waiting).
6 **if** $\tilde{x} \in A_k$ **then**
7 **if** *Local optimization method reports $\tilde{x}$'s run is complete* **then**
8 Add the minimizer from $\tilde{x}$'s run to X_k^*; remove points from $\tilde{x}$'s run from A_k.
9 **else**
10 Query the local optimization method and add the subsequent point (if not already in H_k) from $\tilde{x}$'s run to Q_k.
11 Add $\tilde{x}$ to L_k or S_k, update $H_k = L_k \bigcup S_k$, and update r_k using (2).
12 Start local optimization method at all points in H_k satisfying the conditions Table 1 for μ, ν, and r_k. For each run, add the method's subsequent point (if not already in H_k) to Q_k.
13 Kill runs (remove their points from A_k and any corresponding points from Q_k) with candidate minima within 2ν of each other, keeping the run that was started first.
14 Give w a point x' at which to evaluate f, either from Q_k or $\mathscr{R}_S$. If x' is from Q_k, add it to A_k.
15 Set $X_{k+1}^* = X_k^*$, $H_{k+1} = H_k$, $L_{k+1} = L_k$, $S_{k+1} = S_k$, and $A_{k+1} = A_k$. Increment k.

3.2 Convergence result

Before proving that Algorithm 1 will almost surely identify all minima of (1), we state several assumptions.

Assumption 1 We assume that f is twice continuously differentiable on $\mathscr{D}$, that (1) has no local minima on the boundary of $\mathscr{D}$, that there is a distance $\epsilon_x > 0$ between all local minima of f in $\mathscr{D}$, and that the domain $\mathscr{D}$ is compact.

The compactness of $\mathscr{D}$ and the ϵ_x separation between minima imply that (1) has finitely many minima.

In addition to assumptions on (1), multistart algorithms must make assumptions about the local optimization method. For example, the analysis of MLSL in [27,28] assumes that if the local optimization method is started from a point within a distance ν of a local minimizer, it will recognize (i.e., converge to) that minimizer. BAMLM's assumptions are more explicit about what is required of a local optimization method. Specifically, BAMLM assumes that once the local optimization method is within a distance ν of a local minimum x^* (for ν sufficiently small), the local optimization method will characterize x^* as a minimum in a finite number of function evaluations.

The assumptions made by MLSL and BAMLM on a local optimization method's ability to recognize a local minimum in a finite number of evaluations are not satisfied by many methods, for example, methods with asymptotic convergence results. MLSL brushes away such concerns by assuming that any started local optimization run finds a local minimum before proceeding. In other words, MLSL is implicitly assuming that all started runs require a finite number of evaluations in order to exactly calculate a local minimum. We make no assumption that a local optimization method will stop after a finite number of evaluation.

For the theoretical results of Algorithm 1, we require the following assumptions on the local optimization method.

Assumption 2 The local optimization method used in Algorithm 1 is

(A) strictly descent and converges to a local minimum and
(B) reports its candidate for the local minimum every iteration.

We make an assumption [Assumption 2(B)] that is able to be satisfied by local optimization solvers and is thus more reasonable than the assumptions BAMLM and MLSL make on the local optimization method. Assumption 2(A) is also made by BAMLM and MLSL, though we do not expect such an assumption to be verifiable in practice.

BAMLM and MLSL also assume the local optimization method is *strictly descent* for a path contained in $\mathscr{D}$. That is, starting from any point $x \in \mathscr{D}$, the method must generate some sequence of points $x^{k'}$, with $x^{k'+1} = x^{k'} + p^{k'}$, that converges to a local minimum x^* such that $f(x^{k'} + \beta_1 p^{k'}) \leq f(x^{k'} + \beta_2 p^{k'})$ for all k' and all β_1, β_2 satisfying $0 \leq \beta_1 \leq \beta_2 \leq 1$. That is, of all points evaluated by the local optimization algorithm, there must be a subsequence $\{k'\}$ of iterates so that f is nonincreasing on the line connecting $x^{k'}$ to $x^{k'+1}$. The strict descent assumption can be relaxed [27,28]; although the methods that we employ in our numerical tests do not satisfy this assumption, we use it to guide our development of Algorithm 1.

We also make explicit assumptions about two algorithmic tolerances.

Assumption 3 The parameters ν and μ are sufficiently small so that

(A) for any minimizer x^* in the set of all local minimizers X^*, a local optimization run with a candidate minimum within $\mathscr{B}(x^*, 3\nu)$ will converge only to x^*, and
(B) $\partial_\mu \mathscr{D} \bigcap \mathscr{B}(x^*, \nu) = \emptyset$ for all $x^* \in X^*$ (where $\partial_\mu \mathscr{D}$ is the set of points within a distance μ of the boundary of $\mathscr{D}$).

When Assumption 1 holds, both of the conditions in Assumption 3 are easily satisfiable by choosing ν and μ to be arbitrarily small.

We now show that Algorithm 1 starts finitely many local optimization runs.

Theorem 1 *If f satisfies Assumption 1, the local optimization method satisfies Assumption 2, and ν and μ satisfy Assumption 3, then if Algorithm 1 continues forever, there will almost surely be a finite number of local optimization runs that have a point evaluated (other than their starting point).*

Proof Let $\mathscr{Y}_\nu$ be the points in $\mathscr{D}$ within a distance ν of an element of X^*.

The proof follows similar logic as [21, Theorem 2]. The conditions in Table 1 for starting a run are more restrictive than the original MLSL conditions, (S2)–(S5). Therefore, Algorithm 1 starts fewer runs from points within $\mathscr{D} \backslash \mathscr{Y}_\nu$ than does MLSL (when MLSL samples one random point at each iteration) and MLSL almost surely starts finitely many runs from $\mathscr{D} \backslash \mathscr{Y}_\nu$ [27].

Now consider runs started from points in $\mathscr{Y}_\nu$. In the worst case, for each $x^*_{(i)} \in X^*$, at least one point in $\mathscr{B}(x^*_{(i)}, \nu)$ will be a candidate for starting a run. For each $x^*_{(i)} \in X^*$, it is not possible to have two runs with points evaluated (other than their starting points) within $\mathscr{B}(x^*_{(i)}, \nu)$ because of the killing of runs in Line 13.

Given that X^* is finite, the total number of local optimization runs that have a point evaluated (other than their starting point) in Algorithm 1 is therefore finite almost surely. □

Since our algorithm may start more local optimization runs than the number of possible concurrent evaluations, we must ensure that no run is systematically ignored when workers are given points in Line 14 of Algorithm 1. We therefore assume the following in order to show that every minimum will be identified.

Assumption 4 The procedure for selecting points from Q_k to evaluate satisfies the following: There almost surely exists a natural number $K_0 < \infty$ such that for any K_0 consecutive iterations of Algorithm 1, the probability of taking a point from the sample stream $\mathscr{R}_S$ and from each of the local optimization runs in Q_k is bounded away from zero.

For example, taking a point from $\mathscr{R}_S$ and then cycling through the runs with points in Q_k when giving points to available workers to evaluate is a process that satisfies Assumption 4. This is because Theorem 1 ensures that the number of local optimization runs with a second point evaluated is almost surely finite (and when selecting points from Q_k, only these local optimization points are considered). Another approach for giving points to idle workers is to uniformly choose a point in $Q_k + \{x^s\}$ where x^s

is the next point in $\mathscr{R}_S$. Also note that Assumption 4 is easily satisfied when Q_k is empty.

The following lemma will be useful when showing every local minimum is identified.

Lemma 1 *Let Assumptions 3–4 hold. Then, for every $x^*_{(i)} \in X^*$, Algorithm 1 will evaluate an $x_{(i)} \in \mathscr{B}(x^*_{(i)}, \nu) \bigcap \mathscr{R}_S$ almost surely.*

Proof By Assumption 3(B), $\text{vol}\left(\mathscr{B}(x^*_{(i)}, \nu)\right) > 0$ and $\mathscr{B}(x^*_{(i)}, \nu) \subset \mathscr{D}$. Assumption 4 implies that infinitely many points from $\mathscr{R}_S$ are evaluated. The result follows because $\mathscr{R}_S$ is uniform over $\mathscr{D}$. □

We can now show that Algorithm 1 almost surely identifies all minima of (1).

Theorem 2 *Let Assumptions 1–3 hold, and let the manner in which points are given to workers satisfy Assumption 4. Then, if Algorithm 1 continues forever, each $x^* \in X^*$ will almost surely be calculated in a finite number of evaluations or have a single local optimization run that is converging asymptotically to it.*

Proof For every $x^*_{(i)} \in X^*$ consider the first point $x_{(i)} \in \mathscr{B}(x^*_{(i)}, \nu)$ drawn from $\mathscr{R}_S$. By Lemma 1, such a point will be drawn almost surely. If $x^*_{(i)}$ has not yet been classified as a minimum, then since $\lim_{k\to\infty} r_k = 0$, there is almost surely an iteration $\hat{k}$ where a run could be started from $x_{(i)}$; such a run will be started by the conditions in Table 1 if there does not exist another run with a candidate minimum $\tilde{x} \in \mathscr{B}(x_{(i)}, 2\nu)$. If there were such an $\tilde{x}$, it would be in $\mathscr{B}(x^*_{(i)}, 3\nu)$ and by Assumption 3(A) will converge to $x^*_{(i)}$. If $x^*_{(i)}$ has not been classified as a minimum and no run with such a candidate minimum $\tilde{x}$ exists, then a run will start from $x_{(i)}$ and converge to $x^*_{(i)}$.

Since there are a finite number of runs with points in Q_k at Line 14 of Algorithm 1, Assumption 4 ensures that each run will almost surely progress, and because progress guarantees convergence to a local minimum by Assumption 2(A), the result is shown. □

Therefore, by Theorems 1 and 2, Algorithm 1 almost surely identifies all minima of (1) while starting only finitely many local optimization runs.

4 APOSMM implementation

We now detail APOSMM, an implementation of Algorithm 1. Theoretical results for Algorithm 1 may not hold for this implementation because many local optimization methods return only approximate stationary points (rather than certified minimizers). Therefore, in contrast to Algorithm 1 which seeks high-quality minimizers, in practice the APOSMM implementation generates a set of points with low objective values within an allowed budget. APOSMM was developed in PYTHON using the message-passing interface MPI (accessed via the MPI4PY package [6]). In this implementation, an APOSMM component performs one of three distinct roles: worker, custodian, or manager.

4.1 APOSMM workers

The basic role of an APOSMM worker is to receive a point from the manager, evaluate the objective function at that point, and return the computed objective value (and derivative(s), when available) to the manager. We assume that the cost of evaluating the objective is large enough to outweigh any manager-to-worker communication costs. An APOSMM worker may consist of multiple MPI ranks. This commonly occurs when evaluating the objective involves executing a simulation that uses parallel resources.

4.2 APOSMM custodians

APOSMM custodians are the caretakers of the local optimization runs: They are responsible for generating the next point in a given run. Upon the manager's request, a selected custodian produces the next point in a given local optimization run. The manner in which such points are generated depends on the local optimization method used by the custodian. If the method's state can be efficiently saved and restarted, then the manager must give this information to the custodian. The custodian can then initialize the local optimization method to its previous state and generate the next point in the run. Such state information may not be readily available for many local optimization methods. A simpler, though more computationally intensive approach, is to ensure that the manager provides the custodian with all previously evaluated points from the run for which it must generate the next point. The custodian can then start the method from scratch and sequentially give already-computed function values to the method. When the list of values has been exhausted, the custodian can return the next-requested point to the manager. Any local optimization method can be linked with an APOSMM custodian in this fashion, provided the method generates points deterministically. This approach is used by APOSMM to generate points for local optimization methods in PETSc [2] and NLopt [18].

4.3 APOSMM manager

The APOSMM manager is responsible for maintaining the history of previously evaluated points and a queue of points requested by the custodians. The manager also determines which points need to be evaluated by the APOSMM workers and informs APOSMM custodians regarding which local optimization runs need to be advanced as function evaluations are returned from the workers. After the initialization phase, the manager receives data from any worker or custodian that attempts to contact it.

If a custodian returns a local optimization point, the manager checks the history H_k to determine whether that point has previously been evaluated. If it has, a custodian is restarted with this additional information. If the point requested by the custodian has not been evaluated, the point is added to the queue of local optimization points Q_k. A custodian can also inform the manager that a local optimization run has ruled an iterate to be a local minimum, in which case the manager performs a clean-up operation, marking all points from the run as inactive and adding the run's minimum to X^*.

If a worker contacts the manager with an evaluated point, its function value is stored in the history H_k. If the point is from a local optimization run, a custodian is instructed to generate the next point on this run. When a function value is returned, the manager determines whether a new local optimization run needs to be started (e.g., if r_k is now sufficiently small so that a previously evaluated point should start a run, or if the point that was just evaluated has a promising function value). Idle workers are then given points to evaluate; this action happens immediately unless synchronization is desired, in which case all custodians and workers must be idle before workers are given points. The pseudocode for the APOSMM manager is provided in Algorithm 2.

Algorithm 2: (A)POSMM manager logic

```
Set k = 0, async ∈ {true, false}, and fevalmax.
Initialize history H_k, local optimization queue Q_k, active set A_k, X*_k, and random stream R_S.
Give all workers points to evaluate.
while true do
    MPI.recv(MPI.ANY_SOURCE)
    if Received from custodian then
        if The run has found a minimum then
            Add the minimum to X*_k
            Remove run's points from A_k
        else
            Check H_k and restart custodian if needed.
            Add new point to Q_k.
    if Received from worker then
        Increment k and update H_k.
        if k ≥ fevalmax then break;
        Possibly start a custodian working on the next point.
        Add new points satisfying (S1)–(S5) and (L1)–(L7) from Table 1 to Q_k.
    if async=true OR all workers/custodians are idle then
        Give point(s) from Q_k or R_S to available worker(s).
```

For edification, we diagram the movement of information within APOSMM in Fig. 1. In this example, we start when the objective function value at $\bar{x}$ (a point previously requested by some custodian using some local optimization method A) is received by the manager and is stored in the history. Because the evaluation was from a local optimization run, a custodian is informed that the subsequent point from $\bar{x}$'s run is needed. The custodian produces x', the next point in the local optimization run as determined by the local optimization method A. The custodian relays this information to the manager, which checks the history to determine whether x' has already been evaluated. If it has been previously evaluated, a custodian would be restarted with the function value at x'. In this example, x' has not been evaluated, and so it is added to the queue of local optimization points Q_k. When the manager decides that x' should be evaluated, this point is given to a worker.

Note what does not happen within APOSMM. Workers are indistinguishable: they are not tied to any region of the domain or particular local optimization run. There

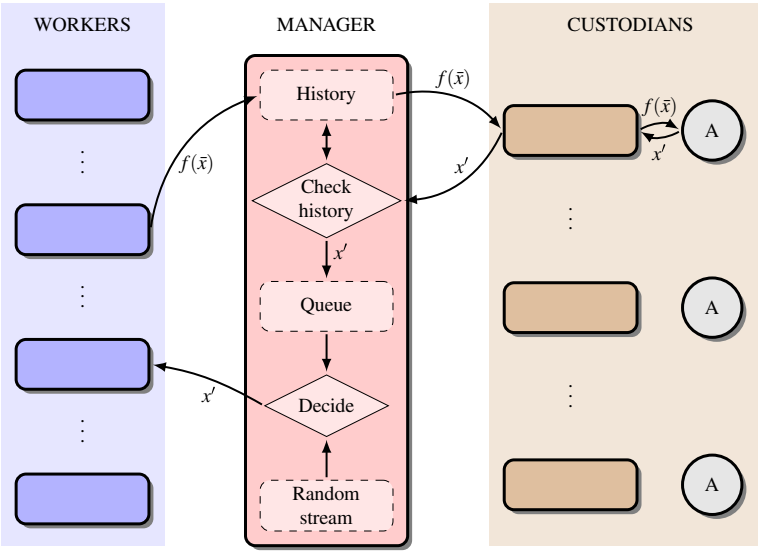


Fig. 1 Diagram of APOSMM following the function value of a local optimization point from a worker to the manager. The manager gives this information to a custodian in order to advance the point's local optimization run. Once a subsequent point has been produced by the local optimization method A, it is added to the manager's queue (provided the new point has not been evaluated)

is no one-to-one mapping of active local optimization runs to APOSMM custodians. In fact, such a mapping would be undesirable because it would require knowing a priori how many (active) runs the manager will start. In this scenario, initializing APOSMM with excessive custodians wastes limited resources as custodians for all incomplete local optimization runs idly await a function evaluation from some worker.

4.4 APOSMM versus POSMM

Two instances of APOSMM using the same random stream $\mathscr{R}_S$ may produce different results, for example, when custodians and workers report to the manager in a different order. At the cost of decreased computational efficiency, we can enforce some level of determinism by ensuring that all workers and custodians are idle (i.e., the manager has received function values from all workers and has received points from any custodian advancing a local optimization run) before giving points to any worker in Line 18 of Algorithm 2. We refer to this modification as POSMM in order to highlight the lack of asynchronicity; POSMM is not necessarily deterministic since the manager does not, for example, query all workers in order. As will be shown in Sect. 5, the sequences of points generated by APOSMM and POSMM for a given random stream $\mathscr{R}_S$ are essentially the same for all problems considered. Since they are similar in many respects, we often use (A)POSMM to represent the collective "APOSMM and POSMM."

4.5 (L8) test

The majority of the work performed by the manager occurs when updating the history after receiving a function value from a worker and when deciding whether (and where) a local optimization run should be started. This effort can be greatly reduced if condition (L8) is not included in the conditions checked by the manager when deciding where to start a local optimization run (as is the default in (A)POSMM).

Condition (L8) is difficult to check in practice since it requires the pairwise distances between points in L_k and all other points. Repeatedly recalculating these distances is onerous; storing, for example, 30,000 double-precision pairwise distances requires over 7 GB of memory.

We consider condition (L8) to be more of a theoretical convenience than a condition that significantly affects an implementation's performance. For nearly all problems tested, inclusion of the test did not affect performance of the implementation in any way: Seldom does a point $\hat{x} \in L_k$ satisfy (S1)–(S5) and (L1)–(L7), but there is no r_k descent path from some point in S_k to $\hat{x}$.

5 Numerical results

We now analyze the numerical performance of various algorithmic implementations—including (A)POSMM, our implementation of Algorithm 1—on a set of problems of the form (1) where $\mathscr{D}$ is a bound-constrained domain. As noted previously, our algorithm and implementation assume nothing about the availability of ∇f. Yet, we find that (A)POSMM's efficient use of previous function evaluations makes it especially well suited for problems where ∇f is unavailable. We therefore compare (A)POSMM exclusively on problems where ∇f is unavailable.

5.1 Synthetic test problems

Because we wish to measure the implementations' abilities to find multiple local minima, we use the GKLS problem generator [9] to construct the set of benchmark problems. These problems have known minima, thus removing ambiguity about X^* and allowing us to accurately measure how an implementation approximates the local minima of each problem after each function evaluation. (Of course, none of the implementations requires knowledge of X^*.) Each GKLS problem is constructed by augmenting a convex quadratic with polynomials, thereby introducing local minima at (known) random locations, while producing a continuously differentiable objective.

One must slightly modify the default GKLS problem generator so that each implementation has a reasonable opportunity to find the global and nonglobal minima. The default generator produces problems where the distance r^* from the minimum of the quadratic to the global minimum satisfies

$$0 < r^* < \frac{1}{2} \min_i \{u_i - l_i\},$$

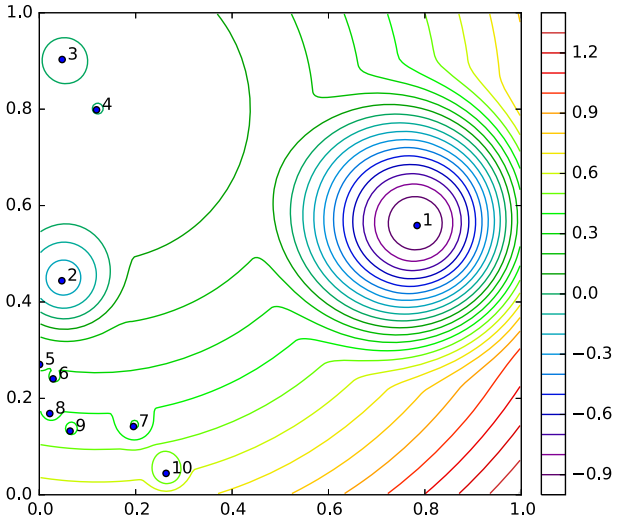


Fig. 2 The first ten-minima GKLS problem in $\mathbb{R}^2$; minima are numbered in order of their function value

where u and l are upper and lower bounds on the problem domain, respectively; it then sets $\frac{r^*}{2}$ as an upper bound on certain level sets of the global minimum. Since we use the unit cube for all problem domains, the volume of such a basin in $\mathbb{R}^7$ (for example) is $\frac{\pi^{7/2}}{\Gamma(\frac{7}{2}+1)}\frac{1}{4}^7 \approx 0.03\%$ of the domain. While such a basin is a tiny portion of the domain in $\mathbb{R}^7$, it is over 20% of the domain in $\mathbb{R}^2$ (provided the basin is contained in $\mathscr{D}$, which is not guaranteed by GKLS). We believe the differences in the percentage of the domain that is included in the basin of attraction is too large for the dimensions considered here. We therefore modify the upper bound on r^* to be $\frac{\sqrt{n}}{2}$ for problems on the n-dimensional unit cube. This increase means that we must ensure that the global minimum is placed within the domain; this is easy to check as the problems are generated. We generate ten such problems each with ten minima in each of the dimensions $n = \{2, \ldots, 7\}$ to obtain 60 problems in total.

An example of the first ten-minima GKLS problem instance in $\mathbb{R}^2$ is shown in Fig. 2. Note that there is no lower bound on the size of the basin of attraction for any local minima. Hence, some minima may be difficult to identify, especially in higher dimensions.

5.2 Data profiles

For a set of implementations M that are run on each problem in a set P, we measure performance using *data profiles* [23]. Data profiles are an absolute metric: they do not change depending on the implementations in M. If an implementation $m \in M$ "solves" problem $p \in P$ in $t_{p,m}$ function evaluations, the data profile of m is

$$d_m(\alpha) = \frac{\left|\left\{p \in P : \frac{t_{p,m}}{n_p+1} \leq \alpha\right\}\right|}{|P|}, \quad \alpha \geq 0, \tag{3}$$

where n_p is the dimension of problem p. In words, $d_m(\alpha)$ is the fraction of problems m solves in a budget of $\alpha(n_p + 1)$ function evaluations. Note that when P consists of problems of varying dimensions, the metric (3) considers the difficulty of a problem to grow linearly with the problem dimension.

5.3 Measuring performance

The essential component underlying any data profile is how the number of function evaluations required to solve a problem, $t_{p,m}$, is calculated. We propose two tests to measure when an implementation has solved a given problem to a given level $\tau \geq 0$.

The first convergence test measures whether function values have been found close to the global minimal value. We consider an implementation to find a global minimum at a level $\tau \in (0, 1)$ after k evaluations if there is a point $x \in H_k$ such that

$$f(x) - f_G \leq (1 - \tau)\left(f(x_c) - f_G\right), \tag{4}$$

where x_c is the centroid of $\mathscr{D}$ and f_G is the value of f at global minima. If an implementation satisfies (4) for a problem p and a given τ, then the implementation found a fraction $(1 - \tau)$ of the best possible decrease from the centroid on problem p.

Defining a test that measures how well an implementation finds a set of j local minima is more difficult. One cannot monitor only the function values observed by an implementation since observations in function space do not provide enough information about the implementation's progress in $\mathscr{D}$ toward minima. We therefore measure the minimum distance from each of the j best local minima to the points evaluated by an implementation.

It can be deceiving to fix a tolerance $\tau > 0$ (independent of the problem dimension) and consider an implementation to have satisfied a convergence test when it has evaluated a point within a distance τ of a set of local minima. This is because it is more difficult to find a point within a τ ball of a local minimizer in higher dimensions: a ball in $\mathbb{R}^n$ with radius 0.1 around a local minimizer has a volume that is approximately 3% of the volume of a unit cube domain when $n = 2$, but 0.00004% of the volume of the unit cube when $n = 7$. We believe that drawing a point uniformly from the domain should have a constant probability (independent of dimension) of satisfying a test of being sufficiently close to a local minimizer.

The distance metric that we proposed in [21] satisfies this requirement. We set

$$\rho_n(\tau) = \sqrt[n]{\frac{\tau \operatorname{vol}(\mathscr{D})\,\Gamma(\frac{n}{2} + 1)}{\pi^{n/2}}}, \tag{5}$$

where $\tau \in (0, 1)$ is some fraction of the domain. This distance ensures that points drawn uniformly from $\mathscr{D}$ are as likely to be within a distance of a minimum in $\mathbb{R}^n$ independent of n. Note that the definition of $\rho_n(\tau)$ depends on the domain $\mathscr{D}$ and therefore depends on how the domain is scaled.

Defining the j best minima is easy when the minima have distinct function values. If duplicate function values exist, however, care must be taken to credit an implemen-

tation for evaluating points close to any of the minima with the same function value as one of the j best. We consider an implementation to find the j best minima after k evaluations if

$$\left|\left\{x^*_{(1)}, \ldots, x^*_{(\underline{j}-1)}\right\} \bigcap \left\{x^*_{(i)} : \exists x \in H_k \text{ with } \left\|x - x^*_{(i)}\right\| \leq \rho_n(\tau)\right\}\right| = \underline{j} - 1$$

and

$$\left|\left\{x^*_{(\underline{j})}, \ldots, x^*_{(\bar{j})}\right\} \bigcap \left\{x^*_{(i)} : \exists x \in H_k \text{ with } \left\|x - x^*_{(i)}\right\| \leq \rho_n(\tau)\right\}\right| \geq j - \underline{j} + 1. \quad (6)$$

For example, if $j = 2$ and the problem has only three minima, each with the same function value, an implementation is considered to solve that problem at a level τ after evaluating points within $\rho_n(\tau)$ of any two of the three minima.

We consider the two convergence criteria (4) and (6) as providing complementary information about an implementation's performance. Note that there is no connection between the convergence tests (4) and (6) for a given value of τ. For example, as we will see, the latter convergence test is much more sensitive to perturbations in τ than is the former.

5.4 Performance of APOSMM and other implementations

We now compare the performances of (A)POSMM, CMA-ES, pVTDIRECT, GLODS, and pMLSL (a parallel version of MLSL) using the convergence tests (4) and (6). We include (serial) DIRECT in the set of benchmarked implementations to show the idealized performance of its class of implementations. RANDOM sampling, drawing points uniformly from the domain, is also included for a baseline comparison. For consistency, the results of RANDOM sampling (and all implementations) will be presented as if it were a serial implementation, even though its performance scales perfectly with increased concurrency; this presentation allows us to comment on how large the increase in concurrency would need to be before RANDOM sampling would be competitive with other implementations.

Conditions (S1)–(S5) and (L1)–(L7) from Table 1 are tested for determining when a point $\hat{x}$ should start a local optimization run within (A)POSMM, provided at least $10n$ random points have been evaluated. BOBYQA [26] is used to advance all local optimization runs within (A)POSMM; its initial trust-region radius is set to

$$\min\left\{r_k, \left\|u - x^0\right\|_\infty, \left\|x^0 - l\right\|_\infty\right\},$$

where $[l, u]$ is the (unit-cube) domain $\mathscr{D}$. The parameter ν is set to 0 so nothing prevents runs from starting close to points that have already been classified as local minima. The parameter μ is set to 10^{-4}, in part to prevent tiny initial trust-region radii and in part because the NLOPT version of BOBYQA moves starting points that are close to the boundary. Although we vary the number of workers, all runs are performed with one custodian (and one manager). These are the default settings for (A)POSMM.

When (A)POSMM adds points to the queue of local optimization points, Q_k, they are given a random priority from [0, 1]. Idle workers are given the highest-priority points from Q_k or are given points from $\mathscr{R}_S$ if Q_k is empty.

Many of the presented data profiles do not include APOSMM because its performance is nearly identical to that of POSMM's when considering only the sequence of points generated by the implementations. Naturally, the wall-clock time between the two is considerable when there is variance in the function evaluation times, as we will show.

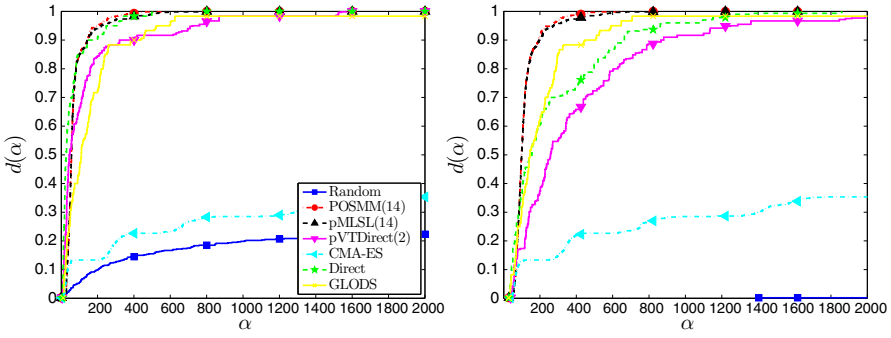
Our implementation of pMLSL differs from POSMM in two ways. pMLSL cycles through local optimization runs, evaluating one point from each active run in the order in which they were started. Also, pMLSL considers only randomly sampled points when deciding where to start new local optimization runs. (Runs are started at points satisfying conditions (S2)–(S5) in Table 1 with r_k defined by (2). (A)POSMM and pMLSL use the same random stream for each problem instance and both use BOBYQA for local optimization runs (with the same settings).

The default settings for the (serial) version of CMA-ES (Version 3.61.b, [12]) that we considered often stops CMA-ES before its budget of evaluations is exhausted. We therefore set the `Restarts` option to a large value so the budget is exhausted. One can increase the number of function evaluations performed in each CMA-ES restart by tightening its (many) default stopping criteria. We avoid doing so because the defaults are appropriately tight (it converges accurately to some minima) and because tightening these tolerances only increases the effort expended in further refining approximate local minima. CMA-ES also has a `EvalParallel` Boolean flag that allows the implementation to query the objective with multiple points. This is not compatible with our setting because the number of points requested by CMA-ES is regularly larger (or significantly smaller) than the specified concurrency. Also, one assumes that the serial version is not less efficient in its use of function evaluations than any parallel implementation is, and (for example) grouping requested points in batches with size equal to the available concurrency would only degrade its performance.

The implementation pVTDirect generates the same iterates independent of the number of workers given. Therefore, we compare only performance (but not timing) using the case where three MPI ranks are given to pVTDirect; rank 0 evaluates the centroid and then coordinates the evaluations performed by the two remaining ranks. Since pVTDirect does not return a history of points evaluated, we monitor its progress including print commands in the objective function that write the point being evaluated, the point's function value, the MPI rank performing the evaluation, and the start and end times of the evaluation. pVTDirect was benchmarked by using similar steps in [14,15].

For GLODS, we use the default parameters.

All implementations were run on each problem with a budget of $2000(n_p+1)$ evaluations, where n_p is the dimension of problem p. The stochastic implementations—(A)POSMM, CMA-ES, GLODS, pMLSL, and Random—were each run with ten different random seeds on each of the 60 GKLS problems to generate 600 runs. (A)POSMM, pMLSL, and Random use the same random stream for each problem instance. The performance of the deterministic Direct and pVTDirect implemen-



(a) $\tau = 10^{-2}$ **(b)** $\tau = 10^{-5}$

Fig. 3 Data profiles using convergence test (4) on 600 GKLS problem instances

tations was replicated ten times for each problem to provide a consistent number of problem instances.

We observe how effectively the implementations find 99% and 99.999% of the possible decrease from the centroid to the global minimum in Fig. 3. Note that the performances of all implementations are being presented on a serial scale; the two concurrent points evaluated by pVTDIRECT and 14 concurrent points evaluated by POSMM and pMLSL are serialized in the order in which they were evaluated. (Because of this serialization of parallel algorithms, it is expected that DIRECT should outperform pVTDIRECT in Fig. 3.)

The abilities of POSMM and pMLSL to approximate the global minimum are nearly identical, though perhaps not using local optimization points delays pMLSL finding of the minimum on a handful of problems.

For many of the higher-dimensional problems in the benchmark set, CMA-ES has difficulty finding the global minimum. For example, for 100 runs on the seven-dimensional problems, all CMA-ES restarts converge to the minimum of the convex quadratic underlying each GKLS problem but never the problem's global minimum. This behavior is possibly due to CMA-ES falsely believing the polynomial augmentations to be noise and instead repeatedly refining the accuracy of a global quadratic model. Nevertheless, CMA-ES does find the global minima for all two-dimensional problems. GLODS and DIRECT both find points with function values that are close to the global minimum, as is shown in Fig. 3b. Finding points close to the global minimum with uniform random sampling (unsurprisingly) performs poorly, as can be seen by RANDOM's data profile. Note that POSMM with two workers would be better than the 14-worker implementation, because the sequence of points produced by 14 workers frequently includes more sample points in the initial iterations than does the 2-worker sequence.

In Fig. 4, we compare the implementations' abilities to find multiple local minima for the benchmark problems. We observe that an implementation's performance as measured by using the convergence criterion (6) is sensitive to the level τ. For example, RANDOM's relative ability to find the three best minima goes from first at a level $\tau = 10^{-2}$ in Fig. 4c to near-last when $\tau = 10^{-4}$ in Fig. 4d. We note that POSMM is consistently effective at finding the j best minima at a level τ for many (j, τ) pairs. We

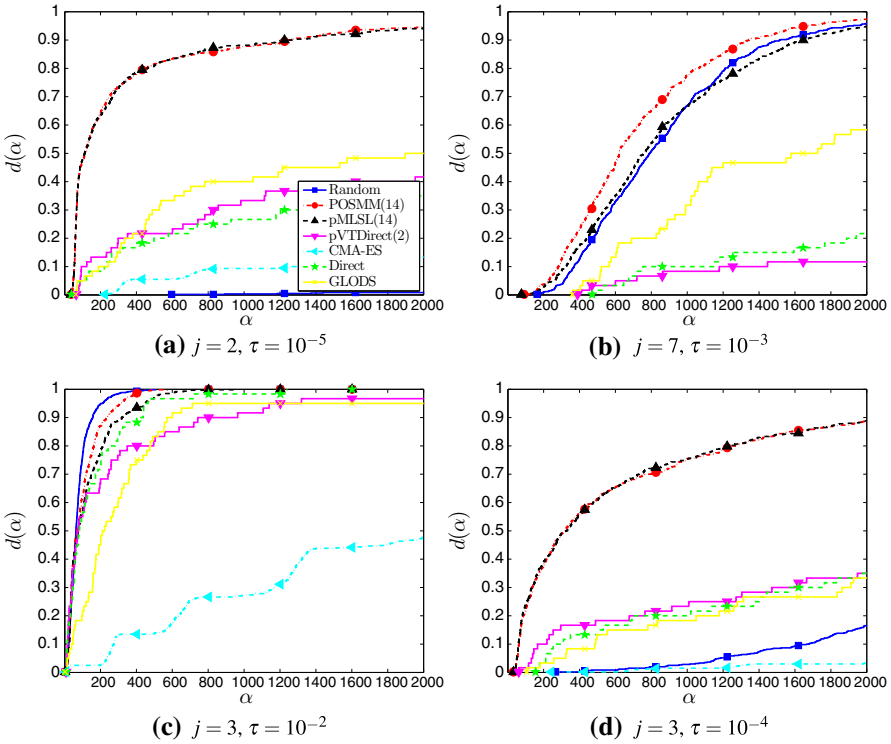


Fig. 4 Data profiles using convergence test (6) on 600 GKLS problem instances

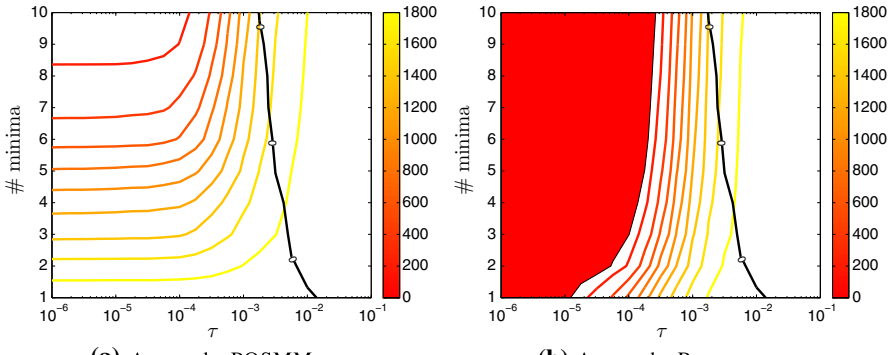
later show that POSMM with 2 workers generates a sequence of points that performs nearly identically to POSMM with 14 workers in terms of the convergence criterion (6).

The most notable difference between POSMM and pMLSL is seen in Fig. 4b. The success of POSMM suggests that using points from local optimization runs is most beneficial when attempting to find more minima to a higher degree of accuracy. GLODS is more successful than the DIRECT implementations, but it is also surpassed by RANDOM.

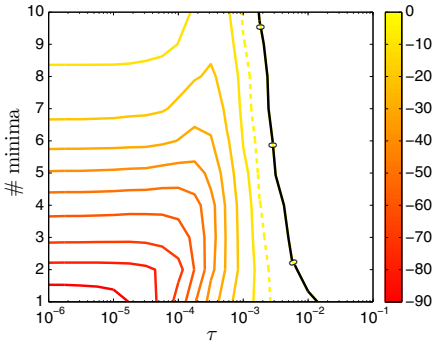
We have observed that the performance of RANDOM sampling with respect to the convergence criterion (4) degrades quickly as the problem dimension increases, suggesting that POSMM will greatly outperform RANDOM sampling in this metric for larger problem dimensions. RANDOM's poor performance remains largely unchanged across problem dimensions with respect to the convergence criterion (6), but this is due to the definition of ρ_n in (5).

5.5 POSMM and RANDOM

The performance of POSMM and RANDOM, especially in Fig. 4b, c may suggest that the success of POSMM is being driven exclusively by its random sampling. We can see from the poor performance of RANDOM in Fig. 4a, d that this is not exclusively the case.



(a) Area under POSMM **(b)** Area under RANDOM

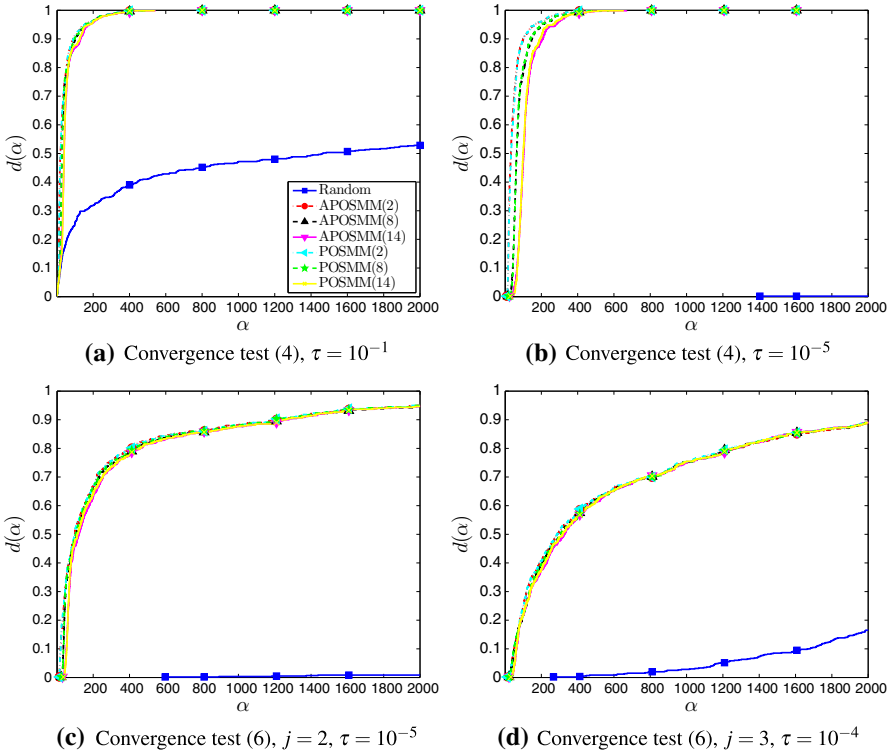


(c) Percent difference

Fig. 5 Contour plots for the areas under the data profile for POSMM and RANDOM for a range of τ and j values. Integral values less than 100 (5% of the maximum possible area) are shaded. Also shown is the percentage difference between the integral of the data profile for RANDOM and the data profile for POSMM, with solid lines at multiples of 10% and dashed lines at -5 and 5%. The approximate line where the percent difference is zero is plotted on all three figures

Nevertheless, we are interested in determining values for j and τ when RANDOM sampling is competitive with POSMM's ability to identify the best minima at a level τ. To this end, we compute the data profiles (3) for POSMM and RANDOM for a range of τ values and for $j \in \{1, \ldots, 10\}$ (because all the problems have ten minima). We then compute the area under each data profile, allowing us to plot contour plots for each (j, τ) pair in Fig. 5a, b. We also plot the difference between RANDOM and POSMM's percentages of the maximum possible area under the data profile (for each j and τ) in Fig. 5c.

Note that the maximum percentage difference in the areas under each data profile is less than 4%, which occurs when $j = 10$ and $\tau = 10^{-2}$ and the area under RANDOM's data profile is 1872 and POSMM's is 1799. (The largest possible value is 2000.) In general, we see that POSMM is significantly better than RANDOM sampling for most values of j and τ tested. The best-case scenario for RANDOM is greatly outweighed by the large range of (j, τ) pairs where RANDOM has a data profile with less than 5% of the possible area.



(a) Convergence test (4), $\tau = 10^{-1}$ **(b)** Convergence test (4), $\tau = 10^{-5}$

(c) Convergence test (6), $j = 2$, $\tau = 10^{-5}$ **(d)** Convergence test (6), $j = 3$, $\tau = 10^{-4}$

Fig. 6 Data profiles for APOSMM and POSMM with different levels of concurrency on 600 GKLS problem instances

5.6 Scalability of APOSMM and POSMM

We now study the scalability (with respect to the number of workers) of the number of function evaluations needed for APOSMM to find (at a level τ) a global minimum or the j best minima. We compare APOSMM and POSMM with different levels of concurrency on the 60 GKLS problems, with each method run on each problem with one of ten random streams. In order to test the possible effects of asynchronous function evaluations, a significant pause (of length drawn uniformly from [0, 0.2] s) is included in each function evaluation. RANDOM sampling is included as a baseline.

In Fig. 6, we measure how effectively (in terms of the number of function evaluations) (A)POSMM approximates minima as the available concurrency changes. Again, there is no scaling by the number of workers on the horizontal axis; only the sequence of points evaluated by each implementation is considered. Figure 6a shows that RANDOM sampling is effective at finding 90% of the possible decrease from the domain centroid to the global minimum on 50% of the problems in $1460(n_p + 1)$ function evaluations. Because RANDOM scales perfectly with increased concurrency, the concurrency would have to be over 140 before RANDOM sampling would be competitive with the $20(n_p + 1)$ function evaluations required for (A)POSMM to solve 50% of

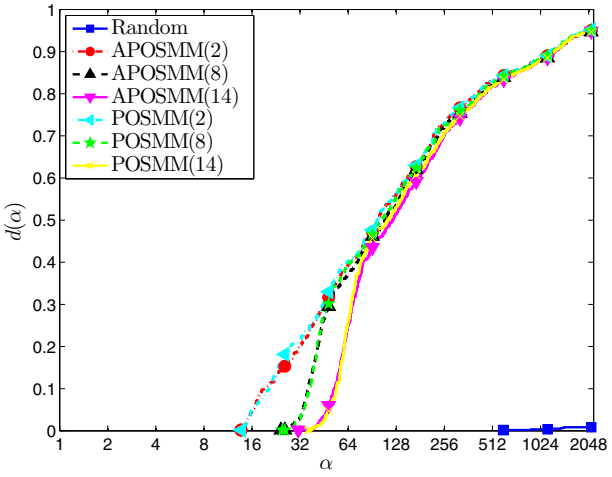


Fig. 7 Data profiles for APOSMM and POSMM with different levels of concurrency using convergence test (6) with $j = 2$ minima at a level $\tau = 10^{-5}$ for 600 GKLS problem instances (Fig. 6c with log scaling)

the problems at this accuracy. That RANDOM sampling solves only one problem to a level $\tau = 10^{-5}$ within $2000(n_p + 1)$ evaluations does not bode well for its ability to accurately approximate a global minimum with reasonable levels of concurrency.

Note that the number of evaluations required to achieve a certain performance level is nearly identical for APOSMM and POSMM instances with the same number of workers. That is, not forcing synchronization on Line 18 of Algorithm 2 does not greatly affect—at the studied levels of concurrency—the order of points generated by our implementation. Increasing the number of workers, and therefore the number of concurrent evaluations, does increase the number of evaluations required to reach a certain performance level, especially in Fig. 6b. Dividing the number of function valuations by the concurrency shows that increasing resources does improve (A)POSMM's ability to approximate a global minimum, but this improvement does not scale perfectly with the additional resources given.

This is not the case with APOSMM's ability to find many minima, as is shown in Fig. 6c, d. Barely any difference exists between the progress toward the j best minima for the sequence of points evaluated by the implementation as the number of workers increases. Such results would be expected if the progress of APOSMM were driven by random sampling alone (since the performance of RANDOM scales perfectly). But RANDOM's poor performance in these figures shows that this is not the case. The results show that APOSMM's ability to find the j best minima scales well with additional resources, at least when the number of minima is relatively similar to the level of concurrency, and that this ability is not due to random sampling alone.

It may seem surprising that increasing the number of workers has relatively little effect on performance of the implementation toward the j best minima, as suggested by Fig. 6c. By plotting the same data on a log scale in Fig. 7, we see that this is largely true, except during the initial iterations.

In Fig. 8, we also plot a histogram of the fraction of local optimization points from each of the 600 POSMM runs on GKLS problems (while accounting for the problem

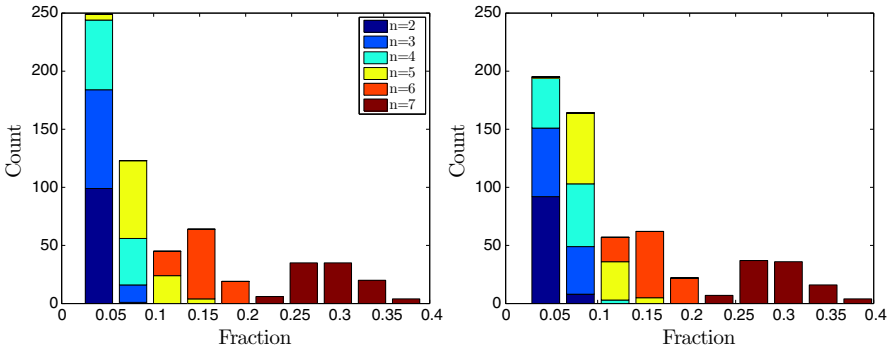


Fig. 8 Histograms of the fraction of local optimization points from each POSMM run on the 600 GKLS problems with 2 workers (left) and 14 workers (right)

dimension). We see that increasing the number of workers increases the fraction of points that are from local optimization runs for only the small-dimension problems.

5.7 Timing scalability

We have shown that APOSMM performs similarly to POSMM in terms of the number of function evaluations required to approximate the j best minima (including $j = 1$); see Fig. 6. This holds even if the cost of evaluating the objective function has high variance. But, of course, the cost of synchronization in Line 18 of Algorithm 2 may incur significant costs in terms of wall-clock time; we now analyze this effect.

Care was taken to ensure that all runs were as equivalent as possible with respect to their timing. Each run occurred on a dedicated 16-CPU node running a minimal operating system, and all function evaluations occurred within RAM that was unique to each node. In order to remove concern about node-to-node communication, the number of concurrent evaluations was limited to 14 (allowing one CPU each for (A)POSMM's manager and custodian). Evaluations were performed in the same manner by each implementation: points of interest were written to RAM, the GKLS executable was called, and function values were read by the implementation. pVTDIRECT's FORTRAN or (A)POSMM's PYTHON may be more or less efficient at such tasks, so we do not draw conclusions between pVTDIRECT and (A)POSMM for a given number of workers. We are more concerned with the implementations' performances as the concurrency changes.

In Fig. 9 we show the wall-clock time required for APOSMM, POSMM, and pVTDIRECT to exhaust their budget of $2000(n_p + 1)$ evaluations on the 600 GKLS instances. Since the function evaluation time is highly variable, POSMM's synchronization before sending points to workers results in poor run-time scaling as the number of workers increases. APOSMM and pVTDIRECT do not incur such synchronization costs, and their median run times scale well with increased workers (for the concurrency levels considered). APOSMM's worst-case wall-clock run time does not appear to scale as well as pVTDIRECT's run time does, although its best-case run time appears to scale slightly better than ideal scaling.

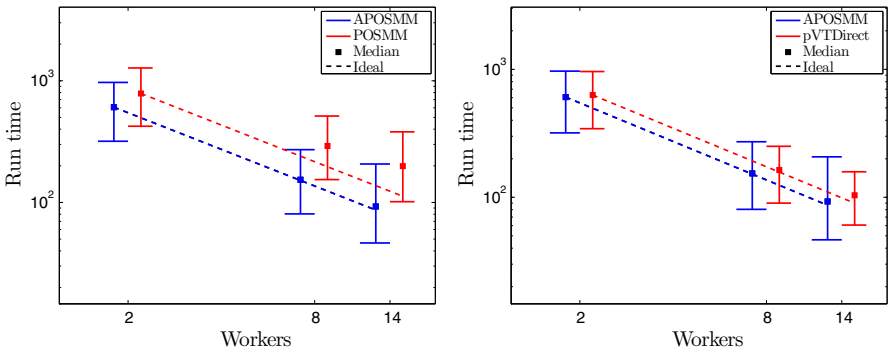


Fig. 9 Time to exhaust $2000(n_p + 1)$ evaluations (log–log scale) for APOSMM versus POSMM (left) and APOSMM versus pVTDIRECT (right) on 600 GKLS problem instances when each function evaluation includes a pause drawn uniformly from $[0, 0.2]$ s. The top whisker is an implementation's worst run time; the bottom whisker is its best run time

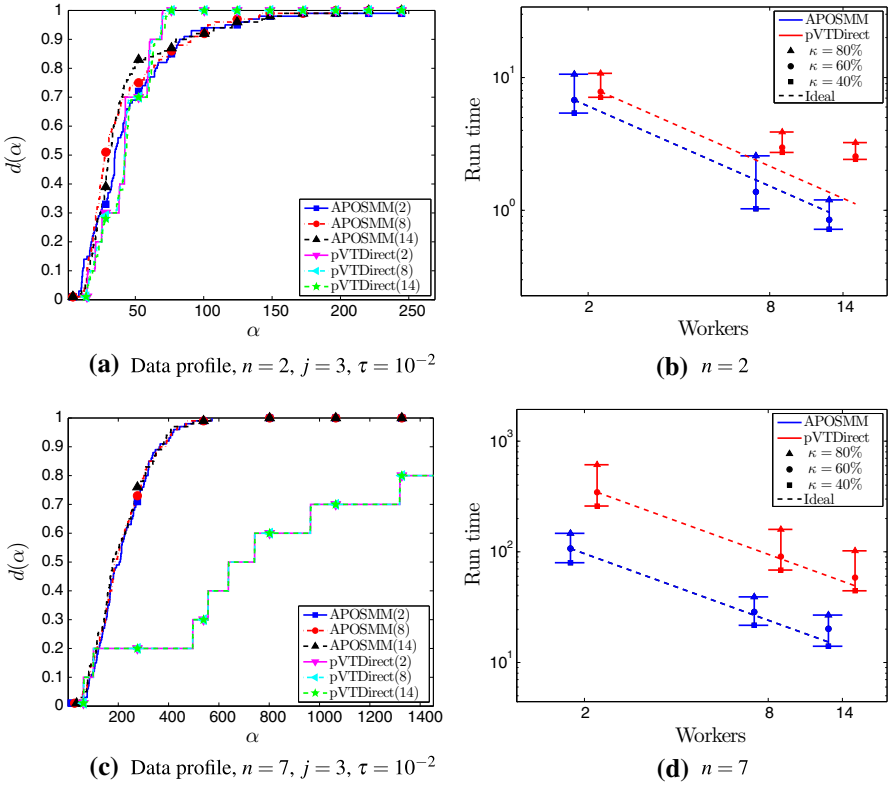
Monitoring an implementation's ability to exhaust a budget of function evaluations (as we do in Fig. 9) does not characterize its performance in terms of wall-clock time. We present results combining an implementation's run time with its performance in different data profiles. Define $C_{p,m}(k)$ to be the cumulative run time from the beginning of the first function evaluation to the end of the kth evaluation. Then the average cumulative run time for a set of problems P is

$$B_m(k) = \frac{1}{|P|} \sum_{p \in P} C_{p,m}(k). \tag{7}$$

We use this quantity to analyze how an implementation's ability (in terms of wall-clock time) to find a global minimum (or the j best minima) to some level τ changes as the concurrency increases.

We remove problem-dimension effects by fixing the problem dimension in a given figure. We plot data profiles for a given (j, τ) combination in convergence test (6). For each implementation m, we define ϕ_m to be the smallest α satisfying $d_m(\alpha) \geq \kappa$. Then we plot $B_m(\phi_m(n_p + 1))$ for a range of workers and κ next to the corresponding data profile. Thus, such plots show the average cumulative run time required for each implementation to have a data profile value larger than κ. Results for $n = 2$ and $n = 7$ are shown in Fig. 10. Note that the relatively conservative values for κ and τ are required since pVTDIRECT does not even find the three best minima at a level $\tau = 10^{-2}$ for more than 80% of the seven-dimensional problems. Of course, pVTDIRECT is not designed to identify multiple minima. The effect of pVTDIRECT generating nearly the same sequence of points independent of the available concurrency can be seen in the nearly identical data profiles for differing numbers of workers in Fig. 10a, c.

We see in Fig. 10b that increasing the concurrency available does not proportionally decrease the amount of wall-clock time required by pVTDIRECT to solve 50% of the $n = 2$ problems. For larger dimensions (Fig. 10d) pVTDIRECT is able to utilize the increased concurrency more efficiently. As the dimension increases from two to seven,



(a) Data profile, $n = 2$, $j = 3$, $\tau = 10^{-2}$ **(b)** $n = 2$

(c) Data profile, $n = 7$, $j = 3$, $\tau = 10^{-2}$ **(d)** $n = 7$

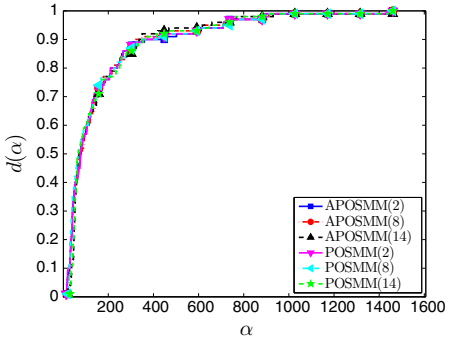
Fig. 10 Data profiles for APOSMM and pVTDIRECT using convergence test (6) and scaling plots for 100 $n = 2$ and 100 $n = 7$ GKLS problem instances

pVTDIRECT requires relatively more wall-clock time to achieve the same level of performance than does APOSMM.

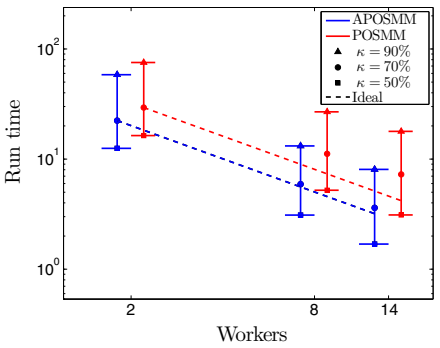
In Fig. 11 we compare time-scaling results for only APOSMM and POSMM for more difficult levels of τ. APOSMM's near-perfect performance scaling is a natural combination of its near-perfect scaling (with respect to time) shown in Fig. 9 and its near-perfect scaling (with respect to finding the three best minima at a level $\tau = 10^{-4}$) shown in Fig. 5d. We consider the scalability of (A)POSMM in terms of run time and performance for relatively tight levels of τ, as is shown in Fig. 11b, d, in order to highlight (A)POSMM's ability to accurately find many minima.

5.8 Forcing runs

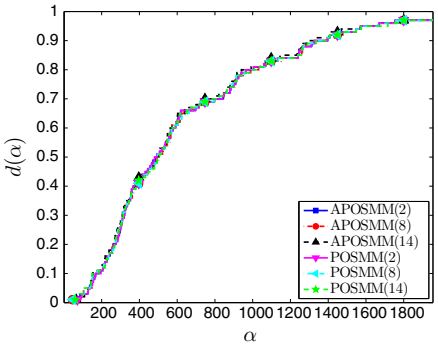
We present preliminary data showing the effects of forcing a different number of runs within POSMM. We require the number of active runs always to be at least 1, 2, 4, or 8 and compare that with the default behavior that places no requirement on the number of active runs. If a lower bound of s runs is desired in the queue Q_k in POSMM but there are not s points that have not started runs or are not in active local optimization runs, all possible starting points will be used to start runs, and any idle workers will



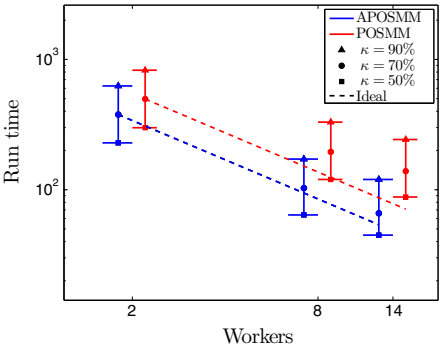
(a) Data profile, $n = 2$, $j = 3$, $\tau = 10^{-3}$



(b) $n = 2$



(c) Data profile, $n = 7$, $j = 3$, $\tau = 10^{-3}$



(d) $n = 7$

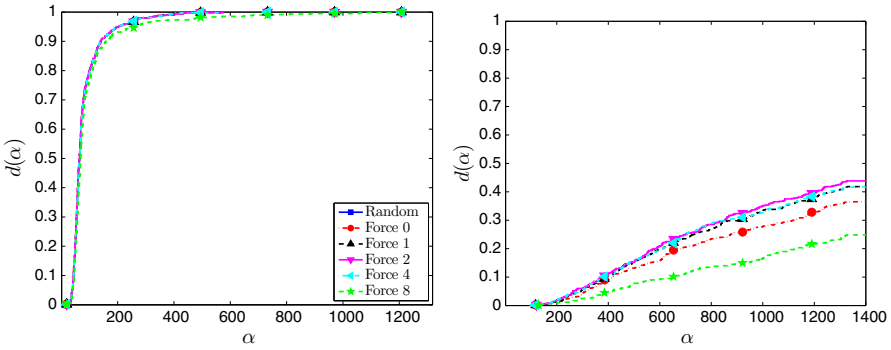
Fig. 11 Data profiles for APOSMM and POSMM using convergence test (6) and scaling plots for 100 $n = 2$ and 100 $n = 7$ GKLS problem instances

be given points to sample. These points will start runs on the next iteration, provided some existing run was not terminated in the interim (we note that we have not observed this in practice).

We observe negligible benefits in the ability to approximate the global minimum [convergence test (4)], but we do observe slight improvements in finding moderate numbers of minima to tighter τ levels, as is shown in Fig. 12. Naturally, this can be overdone if too many runs are forced; forcing eight runs appears to be too many runs for the benchmark problems considered if we are trying to find the six best minima to a level $\tau = 10^{-5}$; see Fig. 12b. (RANDOM does not appear in Fig. 12 or Fig. 13 as it does not satisfy the convergence tests for any of the benchmark problems.)

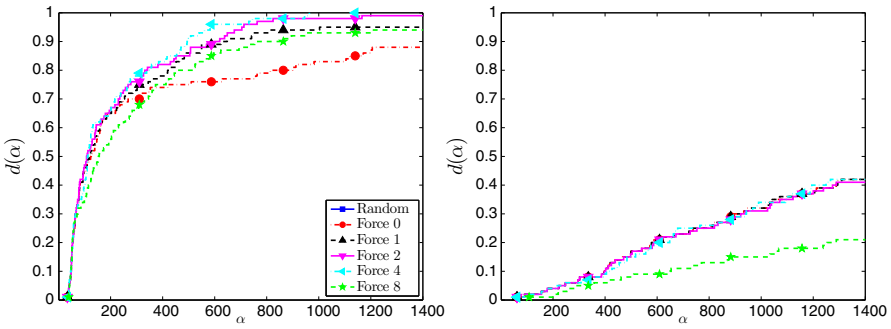
The fact that the relative improvement is low suggests that the rate at which we are decreasing r_k is not limiting POSMM's ability to find minima. Rather, it appears that POSMM has not sampled a point within some of the tiny basins of some minima defined by the GKLS problem generator (see Fig. 2).

Forcing runs does not have a large effect on POSMM's performance with respect to convergence test (4), as is shown in Fig. 12a (and this performance is similar across the problem dimensions considered). With respect to (6), forcing runs appears to help more in smaller dimensions, as is shown in Fig. 13.



(a) Convergence test (4), $\tau = 10^{-5}$ **(b)** Convergence test (6), $j = 6$, $\tau = 10^{-5}$

Fig. 12 Data profiles for 600 GKLS problem instances when forcing a certain number of active runs within POSMM with 8 workers



(a) Convergence test (6), $n = 2$, $j = 3$, $\tau = 10^{-5}$ **(b)** Convergence test (6), $n = 7$, $j = 3$, $\tau = 10^{-5}$

Fig. 13 Data profiles for 100 $n = 2$ and 100 $n = 7$ GKLS problem instances when forcing a certain number of active runs within POSMM with 8 workers

Overall, these results strongly suggests that the rate at which r_k is decreasing is not excessively slow and therefore should not prevent minima from being identified in higher dimensions.

6 Discussion

We have presented an algorithm for identifying multiple high-quality minima of nonlinear optimization problems. Our multistart algorithm considers all previously evaluated points when deciding where to start or continue local optimization runs. Runs are started from points that do not have a better point within a distance r_k.

As more randomly drawn points are evaluated, r_k is decreased. The rate at which r_k is decreased is essential to the algorithm's theoretical properties and practical performance. If r_k decreases too quickly, many redundant runs will be started; if it decreases too slowly, runs that would identify undiscovered minima will not be started. If r_k is decreased at the rate prescribed by (2), we show that under certain assumptions the

algorithm will start only a finite number of local optimization runs but still identify all minima.

Moreover, decreasing r_k by the rate prescribed by (2), the implementation of (A)POSMM performs well in practice for the benchmark problems considered. It is able to find many minima (including a global minimum) quickly, and forcing runs provides only marginal improvement. This result suggests that r_k is decreasing neither too quickly nor too slowly.

We observe that APOSMM's ability to find many minima scales well as the number of concurrent evaluations increases. The asynchronous nature of APOSMM ensures that its performance is not degraded when the function evaluation time has high variance. The libEnsemble library [20] contains an open-source implementation of APOSMM; libEnsemble is summarized in Appendix A.

We are especially interested in finding or developing the best local optimization method for a multistart algorithm such as APOSMM. In practice, it would be useful to have a local optimization method that returns multiple points of interest that help the method converge more quickly to a minimum. We also desire a method that provides a relative value for requested points, thereby helping the APOSMM manager determine the priority in which these points should be allocated to workers. A method that can deterministically be restarted quickly and utilize information from other runs' function evaluations is also a desired property.

Acknowledgements This material was based upon work supported by the U.S. Department of Energy, Office of Science through the Office of Advanced Scientific Computing Research (Contract No. DE-AC02-06CH11357) and the Exascale Computing Project (Contract No. 17-SC-20-SC). We gratefully acknowledge the computing resources provided by the Laboratory Computing Resource Center at Argonne National Laboratory. We thank Gail Pieper for her editing.

A The APOSMM implementation in libEnsemble

The libEnsemble library [20] is designed to manage collections of computations such as those related to concurrent objective function evaluations; libEnsemble includes an implementation of APOSMM. The libEnsemble library coordinates the custodian, manager, and worker functions presented in Fig. 1. By letting libEnsemble handle the coordination of (asynchronous or batch) function evaluations and other calculations, APOSMM can focus on efficiently requesting particular points to be evaluated. This allows flexibility for users wish to customize APOSMM or write their own point-generating and/or point-prioritizing routines.

The sequence of points generated by APOSMM within libEnsemble correspond to the sequence generated by Algorithm 1. When APOSMM is queried by libEnsemble (after prior/initial evaluations have been performed), APOSMM generates points in existing local optimization runs or starts new runs as necessary. If no local optimization runs exist or if insufficient sampling has occurred, then APOSMM produces points drawn uniformly from the specified domain.

The libEnsemble library contains a growing set of example scripts to call APOSMM (and other generating functions) to optimize specified objective functions under different settings. Provided objective examples include simple Python functions as well as

functions based on output from a simulation involving an external MPI-based parallel evaluation. The libEnsemble library also includes example scripts capable of terminating unresponsive simulation-based objectives or observing intermediate simulation output and preempting evaluations. Algorithmic parameters for APOSMM are readily available as part of the examples provided in libEnsemble.

References

1. Audet, C., Dennis Jr., J.E., Le Digabel, S.: Parallel space decomposition of the mesh adaptive direct search algorithm. SIAM J. Optim. **19**(3), 1150–1170 (2008). https://doi.org/10.1137/070707518
2. Balay, S., Abhyankar, S., Adams, M.F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Rupp, K., Smith, B.F., Zampini, S., Zhang, H.: PETSc Web page (2017). http://www.mcs.anl.gov/petsc
3. Besserud, K., Cotten, J.: Architectural genomics, silicon + skin: biological processes and computation. In: Proceedings of the 28th Annual Conference of the Association for Computer Aided Design in Architecture, pp. 978–989 (2008)
4. Cave, R.J., Burke, K., Castner Jr., E.W.: Theoretical investigation of the ground and excited states of Coumarin 151 and Coumarin 120. J. Phys. Chem. A **106**(40), 9294–9305 (2002). https://doi.org/10.1021/jp026071x
5. Custódio, A.L., Madeira, J.F.A.: GLODS: global and local optimization using direct search. J. Glob. Optim. **62**(1), 1–28 (2015). https://doi.org/10.1007/s10898-014-0224-9
6. Dalcín, L., Paz, R., Storti, M., D'Elía, J.: MPI for Python: performance improvements and MPI-2 extensions. J. Parallel Distrib. Comput. **68**(5), 655–662 (2008). https://doi.org/10.1016/j.jpdc.2007.09.005
7. Easterling, D.R., Watson, L.T., Madigan, M.L., Castle, B.S., Trosset, M.W.: Parallel deterministic and stochastic global minimization of functions with very many minima. Comput. Optim. Appl. **57**(2), 469–492 (2014). https://doi.org/10.1007/s10589-013-9592-1
8. García-Palomares, U.M., Rodríguez, J.F.: New sequential and parallel derivative-free algorithms for unconstrained minimization. SIAM J. Optim. **13**(1), 79–96 (2002). https://doi.org/10.1137/S1052623400370606
9. Gaviano, M., Kvasov, D.E., Lera, D., Sergeyev, Y.D.: Algorithm 829: software for generation of classes of test functions with known local and global minima for global optimization. ACM Trans. Math. Softw. **29**(4), 469–480 (2003). https://doi.org/10.1145/962437.962444
10. Gheribi, A.E., Robelin, C., Le Digabel, S., Audet, C., Pelton, A.D.: Calculating all local minima on liquidus surfaces using the FactSage software and databases and the mesh adaptive direct search algorithm. J. Chem. Thermodyn. **43**(9), 1323–1330 (2011). https://doi.org/10.1016/j.jct.2011.03.021
11. Gray, G.A., Kolda, T.G.: Algorithm 856: APPSPACK 4.0: asynchronous parallel pattern search for derivative-free optimization. ACM Trans. Math. Softw. **32**(3), 485–507 (2006). https://doi.org/10.1145/1163641.1163647
12. Hansen, N.: CMA-ES. https://www.lri.fr/~hansen/cmaes_inmatlab.html#matlab. Accessed Nov 2016
13. Hansen, N., Müller, S.D., Koumoutsakos, P.: Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). Evol. Comput. **11**(1), 1–18 (2003). https://doi.org/10.1162/106365603321828970
14. He, J., Verstak, A., Sosonkina, M., Watson, L.: Performance modeling and analysis of a massively parallel DIRECT-Part 2. Int. J. High Perform. Comput. Appl. **23**(1), 29–41 (2009). https://doi.org/10.1177/1094342008098463
15. He, J., Verstak, A., Watson, L., Sosonkina, M.: Performance modeling and analysis of a massively parallel DIRECT-Part 1. Int. J. High Perform. Comput. Appl. **23**(1), 14–28 (2009). https://doi.org/10.1177/1094342008098462
16. He, J., Verstak, A., Watson, L.T., Sosonkina, M.: Design and implementation of a massively parallel version of DIRECT. Comput. Optim. Appl. **40**(2), 217–245 (2007). https://doi.org/10.1007/s10589-007-9092-2
17. Hough, P.D., Kolda, T.G., Torczon, V.J.: Asynchronous parallel pattern search for nonlinear optimization. SIAM J. Sci. Comput. **23**(1), 134–156 (2001). https://doi.org/10.1137/S1064827599365823
18. Johnson, S.G.: The NLopt Nonlinear-Optimization Package. http://ab-initio.mit.edu/nlopt (2017)

19. Jones, D.R., Perttunen, C.D., Stuckman, B.E.: Lipschitzian optimization without the Lipschitz constant. J. Optim. Theory Appl. **79**(1), 157–181 (1993). https://doi.org/10.1007/BF00941892
20. Larson, J.: libEnsemble. https://github.com/Libensemble/libensemble (2017)
21. Larson, J., Wild, S.M.: A batch, derivative-free algorithm for finding multiple local minima. Optim. Eng. **17**(1), 205–228 (2016). https://doi.org/10.1007/s11081-015-9289-7
22. Liuzzi, G., Truemper, K.: Parallelized hybrid optimization methods for nonsmooth problems using NOMAD and linesearch. Comput. Appl. Math. (2017). https://doi.org/10.1007/s40314-017-0505-2
23. Moré, J.J., Wild, S.M.: Benchmarking derivative-free optimization algorithms. SIAM J. Optim. **20**(1), 172–191 (2009). https://doi.org/10.1137/080724083
24. Olsson, P.M.: Methods for Network Optimization and Parallel Derivative-Free Optimization, Ph.D. Thesis. Linköping University. http://liu.diva-portal.org/smash/get/diva2:695431/FULLTEXT02.pdf (2014)
25. Plantenga, T.D.: HOPSPACK 3.0 User Manual, Technical Report October. Sandia National Laboratories, Albuquerque (2009)
26. Powell, M.J.D.: The BOBYQA Algorithm for Bound Constrained Optimization Without Derivatives, Technical Report. DAMTP 2009/NA06, Department of Applied Mathematics and Theoretical Physics, University of Cambridge (2009)
27. Rinnooy Kan, A.H.G., Timmer, G.T.: Stochastic global optimization methods, part I: clustering methods. Math. Program. **39**(1), 27–56 (1987). https://doi.org/10.1007/BF02592070
28. Rinnooy Kan, A.H.G., Timmer, G.T.: Stochastic global optimization methods, part II: multi level methods. Math. Program. **39**(1), 57–78 (1987). https://doi.org/10.1007/BF02592071
29. Ross, S.M.: A First Course in Probability, 8th edn. Prentice Hall, Upper Saddle River (2009)
30. Törn, A., Zilinskas, A.: Global Optimization. Springer, New York (1989). https://doi.org/10.1007/3-540-50871-6
31. Vanden Berghen, F.: CONDOR: A Constrained, Non-linear, Derivative-Free Parallel Optimizer for Continuous, High Computing Load, Noisy Objective Functions, Ph.D. Thesis. Université Libre de Bruxelles. http://www.applied-mathematics.net/optimization/thesis_optimization.pdf (2004)
32. Vaz, A.I.F., Vicente, L.N.: A particle swarm pattern search method for bound constrained global optimization. J. Glob. Optim. **39**(2), 197–219 (2007). https://doi.org/10.1007/s10898-007-9133-5
33. Wild, S.M.: Derivative-Free Optimization Algorithms for Computationally Expensive Functions, Ph.D. Thesis. Cornell University. http://ecommons.cornell.edu/handle/1813/11248 (2009)