MANAGING COMPUTATIONALLY EXPENSIVE BLACKBOX MULTIOBJECTIVE OPTIMIZATION PROBLEMS WITH LIBENSEMBLE

Tyler H. Chang

Dept. of Computer Science
Virginia Polytechnic Institute & State Univ.
thchang@vt.edu

Jeffrey Larson

Mathematics and Computer Science Division
Argonne National Laboratory
jmlarson@anl.gov

Layne T. Watson

Depts. of Computer Science, Mathematics,
and Aerospace and Ocean Engineering
Virginia Polytechnic Institute & State Univ.
ltw@cs.vt.edu

Thomas C. H. Lux

Dept. Computer Science
Virginia Polytechnic Institute & State Univ.
tchlux@vt.edu

ABSTRACT

Multiobjective optimization problems (MOPs) are common across many science and engineering fields. A multiobjective optimization algorithm (MOA) seeks to provide an approximation to the tradeoff surface between multiple, possibly conflicting, objectives. Many MOPs are the result of objective functions that require the evaluation of a computationally expensive numerical simulation. Solving these large and complex problems requires efficient coordination between the MOA and the computationally expensive cost functions. In this work, a recently proposed MOA is integrated into the `libEnsemble` software library, which coordinates extreme scale resources for large ensemble computations. Efficient integration requires fundamental changes to the underlying MOA. The convergence and performance results for the integrated and original MOA are compared on a set of benchmark problems.

Keywords: multiobjective optimization, blackbox optimization, HPC resource managers

1 INTRODUCTION

Multiobjective optimization problems (MOPs) appear in many science and engineering fields, including problems in engineering design optimization (Campana et al. 2018). Whereas single-objective optimization algorithms typically search for a single local or global minimum of a scalar-valued cost function, multiobjective optimization algorithms (MOAs) seek a set of solutions that describe the tradeoff between the problem objectives. Ultimately, this allows a decision maker with domain knowledge and unstated preferences to make an informed decision.

In general, a MOP is defined by a vector-valued cost function $F : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X} \subseteq \mathbb{R}^d$ and $\mathcal{Y} \subseteq \mathbb{R}^p$. Here, $\mathcal{X}$ denotes the *feasible design space* with dimension d, and $\mathcal{Y}$ denotes the *feasible objective space* with dimension p. Conceptually, F can be decomposed into p scalar cost functions $f_i : \mathbb{R}^d \to \mathbb{R}$, $i = 1, \ldots, p$, such that $F(x) = (f_1(x), \ldots, f_p(x))^T$. This paper considers MOPs in a standard form where all component

functions of F are to be minimized. For $Y_1, Y_2 \in \mathcal{Y}$, Y_1 *dominates* Y_2 if Y_1 is componentwise less than or equal to Y_2 with strict inequality in at least one component. A solution $F(x^*)$ is said to be *nondominated* if $F(x)$ does not dominate $F(x^*)$ for all $x \in \mathcal{X}$. If $F(x^*)$ is nondominated, then x^* is said to be *efficient* and the pair $(x^*, F(x^*))$ is *Pareto optimal*. The solution to a MOP is the set of all nondominated points (called the *Pareto front*) and the corresponding *efficient set*. MOAs attempt to find a high-fidelity approximation to the Pareto front and efficient set. This approximation typically consists of a discrete set of approximately Pareto optimal pairs. Further reading on MOPs can be found in the book by Ehrgott (2005).

The multiobjective function F is said to be a *blackbox* when there is no additional information (such as gradients) for any of the component functions f_i other than their values. Often, blackbox functions are the result of computationally expensive numerical simulations. Solving such problems requires the use of large computational resources (Kodiyalam et al. 2004) and often involves an ensemble of codes, including the numerical simulations from which the cost functions are derived and the optimization software. Solving such problems at scale on modern HPC systems can be facilitated by a computational resource manager.

`libEnsemble` is a library for coordinating the concurrent evaluation of dynamic ensembles of calculations (Hudson et al. 2019). The library is developed at Argonne National Laboratory as a part of the DOE Exascale Computing Project; it is designed to use massively parallel resources to accelerate the solution of design, decision, and inference problems and to expand the class of problems that can benefit from increased concurrency levels. `libEnsemble` employs a manager-worker scheme that is controlled by user-defined simulation, generation, and allocation functions (each of which could be using parallel computing resources). The generation function produces parameter values to be evaluated by the simulation function; the simulation function uses the specified parameters to run a (potentially expensive) numerical calculation; and the allocation function decides when a simulation or generation function should be called and with what resources. If one is using `libEnsemble` to solve MOPs, the generation function is a MOA, the simulation function evaluates the blackbox function F, and the allocation function calls the generation function when the simulation function has evaluated all of the previously requested set of parameter values.

This paper studies the performance of two implementations of the recently published MOA of Deshpande, Watson, and Canfield (2016): one implementation uses its own communication framework, and one uses `libEnsemble`. Section 2 provides further background on `libEnsemble` and MOPs, then summarizes the algorithmic details of the MOA that is implemented herein. Section 3 describes both a standalone parallel driver for the MOA and an alternative interface where the MOA is integrated into `libEnsemble`. Section 4 shows the outcome of several experiments, where problems with known Pareto sets are used to analyze the performance of both implementations in terms of convergence to the complete Pareto front, machine utilization, and computational overhead. Section 5 briefly discusses the results of Section 4.

2 BACKGROUND

This section gives greater detail on `libEnsemble`; background is provided on common techniques for solving computationally expensive blackbox MOPs, and the specific MOA of interest is introduced.

2.1 libEnsemble

`libEnsemble` employs a manager process to allocate work to multiple worker processes. A `libEnsemble` worker is the smallest indivisible unit that can perform calculations. For some cases, a worker can be given a single core to generate a sample of points; in other cases, a worker can be given many nodes to perform a complex numerical simulation. While one can allocate different resources to different workers throughout a given `libEnsemble` run, all instances in this paper give each worker equal computational resources. The `libEnsemble` manager-worker communications can utilize various communication media, including MPI,

multiprocessing, and TCP. Interfacing with user-provided executables is also supported; in this paper, the generation function requires a Fortan executable. Each worker can control and monitor any level of work, from small subnode jobs to huge many-node simulations.

In this paper, a persistent generation function is used to produce points to be evaluated: the generation function is initiated at the beginning of the `libEnsemble` run when it provides a batch of points to be evaluated. Only after points in a batch have been evaluated is the generation function informed of the function values so that it can produce the next batch of points. Hence, workers (and their computational resources) will be idle when a worker completes a given function evaluation and all remaining points in the batch are currently being evaluated.

2.2 Common Techniques in Blackbox Multiobjective Optimization

Three broad approaches are used for solving blackbox MOPs. Not all methods neatly fall into one of these categories, but most widely used software packages use one or more of these strategies. The first class consists of multiobjective evolutionary algorithms, such as `NSGA-II` (Deb et al. 2002a). While widely popular, evolutionary algorithms frequently require many function evaluations in each iteration and are generally not practical when F is expensive to evaluate. The second class contains direct multisearch methods, such as the `BiMADS` algorithm from the `NOMAD` software package (Le Digabel 2011). While efficient for problems with few objectives, many of these techniques do not scale well for problems where p is large. In particular, `BiMADS` applies exclusively to the biobjective case.

The third class consists of scalarization techniques, which reduce MOPs to single-objective problems by composing a scalarization function $G : \mathbb{R}^p \to \mathbb{R}$ with F. The resulting function $G \circ F$ can be minimized by using a single-objective blackbox optimization solver to approximate a single nondominated point. Optimizing different scalarizations seeks to produce a set of nondominated points that approximates the Pareto front. One common scalarization scheme is the weighted-sum method, which uses a vector of strictly positive weights $w = (w_1, \ldots, w_p)$ to produce a "weighted average" cost function

$$G_w\big(F(x)\big) = \sum_{i=1}^{p} w_i f_i(x). \tag{1}$$

Because these methods often solve problems with many different weights, modern scalarization techniques often employ some variation of the response surface methodology (RSM). In the context of MOPs, RSM approaches fit a computationally cheap surrogate model $\hat{f}_i$ to **each** component function f_i using designed experiments. Then numerous scalarizations of $\hat{f}_1, \ldots, \hat{f}_p$ can be solved for a relatively minimal cost (Myers, Montgomery, and Anderson-Cook 2016). Algorithm 1 demonstrates how RSM can be combined with the weighted-sum method to solve a computationally expensive MOP. While this scheme is cost effective, naïve scalarization strategies (e.g., drawing w uniformly from $[0, 1]^p$) can produce clustered points in the solution set that yield a poor understanding of the Pareto front's shape. In the context of the weighted-sum method, an *adaptive weighting scheme* is needed in order to select sets of weights that yield well-spaced points.

Algorithm 1: Applies RSM to MOPs with the weighted-sum method.

inputs
$F : \mathbb{R}^d \to \mathbb{R}^p$ is the objective function;
ℓ is the number of scalarizations to be solved;
$\{w^{(1)}, \ldots, w^{(\ell)}\}$ is a set of predetermined scalarizing weight vectors;

begin
Explore the design space (often via a space-filling design, but could be any global search strategy);
Use data from the exploration phase to construct $\hat{F} \leftarrow (\hat{f}_1, \ldots, \hat{f}_p)$ where $\hat{f}_1 \approx f_1, \ldots, \hat{f}_p \approx f_p$;

for $i = 1, \ldots, \ell$ **do**
$\hat{x}^{(i)} \leftarrow \underset{x}{\arg\min}\, G_{w^{(i)}}\left(\hat{F}(x)\right)$;
enddo
Evaluate the candidate designs $\hat{x}^{(1)}, \ldots, \hat{x}^{(\ell)}$;
end

In Algorithm 1, true function evaluations are required only during the exploration phase and when evaluating the candidate designs. In this way, one can predict the solution to numerous scalarized subproblems for little more than the cost of a single design-space exploration.

2.3 The Multiobjective Optimization Algorithm

The MOA of Deshpande, Watson, and Canfield (2016) solves blackbox MOPs subject to simple bound constraints when the component functions are Lipschitz continuous. This algorithm applies RSM in a sequence of local trust regions (LTRs) using batches of adaptive weights. These LTRs are centered on *isolated points* from the current set of nondominated F values, which are determined by projecting the nondominated values into a $(p-1)$-dimensional space and computing their Delaunay graph. For repeated centers, the LTR radius is decayed down to some tolerance, and the LTR center is determined by choosing the most isolated point whose radius would be above the tolerance. In the case where all isolated points would result in a LTR whose radius is below the tolerance, then the problem has been solved to the maximum precision. After a suitable isolated point has been identified, a "square" LTR is constructed in the design space. The Delaunay graph is also used to assign the adaptive weights. For additional details on using the Delaunay graph to determine point isolation and choosing the LTR radius, see the paper by Deshpande, Watson, and Canfield (2016). Algorithm 2 shows how the LTRs are combined with RSM and the adaptive weighting scheme in order to solve a nonconvex MOP, subject to simple bound constraints.

Algorithm 2: Solves a MOP using RSM, adaptive weighting, and LTRs.

inputs
$F : \mathbb{R}^d \to \mathbb{R}^p$: the objective function;
L, U: lower and upper simple bound constraints on the design space;
M: the budget for evaluations of F;
variables
B: a database containing every design point evaluated and its corresponding objective value;
$P^{(k)}$: the set of nondominated points at the start of iteration k;
$D^{(k)}$: the set of efficient points corresponding to $P^{(k)}$;
$\Delta^{(k)}$: the kth LTR with $\tilde{x}^{(k)}$ as its center
$\ell^{(k)}$: the number of Delaunay neighbors of $F(\tilde{x}^{(k)})$ in the projective space;
$\{w^{(1,k)}, \ldots, w^{(\ell^{(k)},k)}\}$: the kth set of adaptive weight vectors;
begin
$B \leftarrow \emptyset$; $k \leftarrow 0$;
begin the 0th (pre) iteration
Statically assign $p+1$ weight vectors for the 0th iteration: $w^{(1,0)}$, ..., $w^{(p+1,0)}$;
Minimize $p+1$ instances of (1) for $w^{(1,0)}, \ldots, w^{(p+1,0)}$ within $[L, U]$ using RSM (Algorithm 1);
Store all evaluations of F in B;
end the 0th iteration
$k \leftarrow k+1$;
while $|B| < M$ **do**
Compute $P^{(k)}$ and $D^{(k)}$ based on the current contents of B;

Identify an isolated point $F(\tilde{x}^{(k)})$ in $P^{(k)}$;
if no isolated point was found **then**
return $P^{(k)}$ and $D^{(k)}$
else
Compute $\Delta^{(k)}$ centered at $\tilde{x}^{(k)}$;
Compute the kth set of adaptive weight vectors $\{w^{(1,k)}, \ldots, w^{(\ell^{(k)},k)}\}$;
Minimize (1) for $w^{(1,k)}, \ldots, w^{(\ell^{(k)},k)}$ within $\Delta^{(k)}$ using RSM (Algorithm 1);
Store all evaluations of F in B;
$k \leftarrow k + 1$;
endif
enddo
return $P^{(k)}$ and $D^{(k)}$
end

The key contribution of Deshpande et al. is the usage of the Delaunay graph to identify isolated points, but Deshpande et al. also provide a novel strategy for RSM based on an adaptive Dividing Rectangles (DIRECT) search. In each RSM exploration phase, DIRECT is applied to each component function f_1, ..., f_p, either within the current LTR or over the entire design space (in the 0th iteration). In the 0th iteration, DIRECT is also run one additional time with an equal weighting of all objectives. After completing the RSM search, a linear Shepard model (Thacker et al. 2010) is used to construct the surrogate models $\hat{f}_1$, ..., $\hat{f}_p$, and these surrogate models are optimized for each of the adaptive weights by using the GPS MADS algorithm (Le Digabel 2011).

For this paper, the DIRECT search is implemented using the `VTDIRECT95` software package (He, Watson, and Sosonkina 2009), and surrogates are implemented with the `LINEAR_SHEPARD` module from `SHEPPACK` (Thacker et al. 2010). Rather than using the widely distributed `NOMAD` software package (Le Digabel 2011), a custom lightweight Fortran implementation of GPS MADS is used because `NOMAD` includes significant extra overhead and machinery that is relevant only for optimizing an expensive blackbox function and the linear Shepard's models are computationally cheap surrogates. An implementation of the scalable Delaunay interpolation algorithm of Chang et al. (2018) is used to compute the Delaunay graph in arbitrary dimension.

3 PARALLEL IMPLEMENTATIONS

In this section, several novel parallel implementations of Algorithm 2 are proposed. The first implementation introduces no significant modifications to the underlying algorithm or RSM strategy, as proposed by Deshpande, Watson, and Canfield (2016). The remaining implementations are tailored to use the `libEnsemble` framework to efficiently exploit parallel resources. In all cases, the cost of evaluating F is assumed to overwhelm all iteration costs. Therefore, the focus is placed on parallelism between independent evaluations of F, and opportunities for parallelizing iteration tasks are not considered. In a real-world application, the evaluation of F may be its own parallel process, featuring concurrency within the implementation of each component function f_i and between evaluations of the component functions f_1, ..., f_p. Since this level of parallelism is specific to the application, it should be implemented by the user. This is explicitly supported by `libEnsemble`, which allows for each simulation function to utilize parallel resources.

3.1 A Parallel Implementation of the Original Algorithm

First, consider a parallel implementation of Algorithm 2, without leveraging `libEnsemble`. Algorithm 1 is called once per iteration of Algorithm 2; and, as previously discussed, there are two locations in Algorithm 1 where F must be evaluated. The second of these locations is trivial to parallelize, since it involves evaluation

over a batch of precomputed design points. However, proper parallelization of the first location (design space/LTR exploration using either $p + 1$ or p instances of `VTDIRECT95`) is slightly more challenging and requires exploitation of two nested levels of parallelism.

In order to efficiently parallelize up to $p + 1$ instances of the DIRECT search, parallelism within each instance of `VTDIRECT95` should be exploited as well as parallelism over multiple calls to `VTDIRECT95`. For parallelism within a single call, `VTDIRECT95` provides a parallel driver subroutine `pVTdirect`, which distributes function evaluations and memory burden over a network using MPI in a fully distributed paradigm, with decentralized memory. Because Algorithm 2 maintains all function evaluation data in a central database B, this fully distributed paradigm is not appropriate for the RSM exploration phase in Algorithm 1. Instead, a slight modification is made to the serial driver `VTdirect` to parallelize its loop over each small batch of "potentially optimal" boxes. Hereafter, this implementation is referred to as `bVTdirect`.

When parallelizing over multiple instances of `bVTdirect`, additional issues arise. Specifically, because the DIRECT algorithm samples on an implicit mesh, multiple DIRECT instances often request the same design points. In order to prevent unnecessary evaluations of F, a system of locks is placed on the database B from Algorithm 2. When a new design point is evaluated, B is first checked to see whether that point has been (or is currently being) evaluated. If so, that instance of `bVTdirect` must wait (if necessary) until the evaluation is complete, then receive the result.

This parallelism strategy is implemented entirely by using OpenMP for shared-memory parallelism. The choice of OpenMP places the burden of across-node distribution on the user, whose implementation of F should involve machinery to distribute F across a network. This paradigm is appropriate for many modern HPC systems because it allows users greater control over how memory and computations are distributed. Because certain OpenMP threads will have to "wait" on design points that are currently being evaluated, there can be many idle OpenMP threads. Generally, it is recommended to overload the master node with OpenMP threads to achieve the maximum number of evaluations. However, only a controlled number of distributed-memory tasks (matching the number of nodes on the network) can be achieved by the user.

3.2 Integration with libEnsemble

Algorithm 2 can also be implemented as a `libEnsemble` generation function. Naïvely, this could be achieved by running Algorithm 2 until evaluations of F are needed (during a reference to Algorithm 1), using `libEnsemble` to distribute the appropriate simulations, recording the results of these simulations in the database B, and proceeding with Algorithm 2. However, one of the goals of `libEnsemble` is to make efficient usage of extreme scale resources. While Algorithms 1 and 2 ultimately evaluate F at small batches of points, there is no mechanism to control the size of each batch, thus leading to poor load balancing when implemented over statically allocated HPC resources. Therefore, two significant modifications to Algorithms 1 and 2 have been made for integration with `libEnsemble` to allow for greater control over the number of concurrent function evaluations.

The first modification is made to the RSM search strategy in Algorithm 1. The DIRECT-based search algorithm proposed by Deshpande et al. is not appropriate for integration with `libEnsemble` for several reasons. First, `libEnsemble` requires decoupling between the optimization algorithm and its calls to F since these tasks are separately controlled by using the simulation and generation functions. The `VTDIRECT95` driver subroutines strongly couple objective function evaluations with the optimization algorithm, so integration of `VTDIRECT95` with `libEnsemble` would require fundamental reworking of the driver subroutines. Second, `VTDIRECT95` sequentially evaluates batches of design points of varying sizes, with no consideration for the amount of available resources. Such behavior makes inefficient use of `libEnsemble`'s leveraging of computational resources.

Suppose that Algorithm 2 is implemented as a `libEnsemble` generator on an HPC system with sufficient resources for n_b concurrent evaluations of F. When n_b is large, an efficient strategy would be to evaluate fewer batches, each containing more design points. Preferably, users should be able to adjust these batch sizes, targeting a multiple of n_b. Therefore, a space-filling design of controllable size is preferable to the DIRECT-based search. So, when `libEnsemble` reaches the RSM search phase of Algorithm 1, the `libEnsemble` generator returns a Latin hypercube design over the current search space. When the size of the Latin hypercube design is small relative to n_b, the size should typically be a multiple of n_b. After the design has been generated, `libEnsemble` coordinates the concurrent simulation evaluations.

The second modification allows users to control the batch size for generating candidate designs at the end of Algorithm 1 in order to match n_b. The adaptive weighting scheme proposed by Deshpande, Watson, and Canfield (2016) produces variable-sized batches of candidate designs by default, which is not conducive to load balancing. Every candidate design for one of the adaptive weightings must be evaluated, in order to ensure that new solutions are found surrounding the current isolated point. So, the goal of this modification is to achieve the smallest possible multiple of n_b that is greater than the original number of candidate designs.

Suppose that n_a additional candidates are needed to achieve a multiple of n_b. In order to find n_a additional candidates to pad out the current batch, a large number (greater than n_a) of **additional** convex weight vectors are randomly generated during each iteration of Algorithm 2. When optimized over the surrogate models, these random weight vectors produce additional candidate designs within the current LTR, in the neighborhood of the current isolated point in the objective space. It is not always possible to find n_a additional candidates, however, especially when n_b is large. The reason is that different weight vectors can yield the same candidate solutions (particularly when F is nonconvex). Therefore, a fixed number of additional random weights is always generated and used to produce a pool of additional candidates. If this pool of additional candidates is larger than n_a, then the first n_a candidates are added to pad out the batch returned by the `libEnsemble` generator function. If fewer than n_a additional candidates are generated, then all additional candidates are added, padding out the batch as much as possible.

Note that generating this additional pool of candidates to pad out each batch significantly increases the iteration costs for Algorithm 2 since many additional surrogate optimization problems must be solved. In order to demonstrate the costs and benefits of this approach, two variations have been implemented in `libEnsemble`. In the first implementation (`libEnsemble1`), no additional candidate solutions are generated. In the second implementation (`libEnsemble2`), additional candidate solutions are generated, as described in the preceding paragraph.

4 EXPERIMENTS AND RESULTS

Two test problems have been run on the Bebop system at Argonne National Laboratory using the implementations described in Section 3. Performance data has been collected for all implementations, showing the degree to which the solutions approximate the Pareto front and the utilization of available system resources.

4.1 Description of Test Problems

Let $d > p$, and let e_i denote the ith standard basis vector in $\mathbb{R}^d$. Then the first test problem is defined by

$$F_c(x) = \left(\|x - \frac{1}{2}e_1\|_2^2, \ldots, \|x - \frac{1}{2}e_p\|_2^2 \right), \quad x \in [-1, 1]^d. \tag{2}$$

The Pareto front for F_c is a portion of a rotated parabola in $\mathbb{R}^p$. Because this problem is convex, it is an easier problem for MOAs. The second test problem is the nonconvex problem `DTLZ2`, as described by Deb et al. (2002b). Its Pareto front is the unit sphere in the positive orthant. This Pareto front is concave, which generally presents a problem for adaptive weighting schemes.

In order to emulate the expense of a blackbox simulation problem, an artificial runtime is simulated using the CPU clock. For both F_c and `DTLZ2`, two variations are created. In the first variation, the runtime is set to one second; in the second variation, the runtime is a random value drawn uniformly from the interval $[0.5s, 1.5s]$. This tests the methods for their ability to load balance function evaluation times that vary.

For all problems considered, the design dimension is $d = 5$. The objective dimensions considered are $p = 2$, $p = 3$, and $p = 4$, and the function evaluation budgets, M, of sizes 1,000, 1,500, and 2,000 are considered. Note that these values of M are an order of magnitude less than the recommendations of Deb et al. (2002b). However, for a computationally expensive numerical simulation (such as a three-dimensional fluid dynamics simulation), these budgets would be considered generous.

Evaluation of how well points evaluated by a MOA approximate the true Pareto front is an open problem (Audet et al. 2018). Three criteria to measure the quality of an approximate Pareto surface are considered:

- The number of nondominated solution points identified
- The distance between points on the approximate and true Pareto front
- The degree to which those points are spread evenly across the entire Pareto front

In this paper, the cardinality, the root mean squared error (RMSE), and the *star discrepancy* as described by Kugele, Trosset, and Watson (2008) of the solution set are used to measure these three criteria, respectively. Note that the RMSE can be easily computed because the test functions have analytic Pareto fronts. In order to compute the star discrepancy, unions of path-connected Delaunay simplices are used as a family of Lebesgue measurable sets. The Delaunay discrepancy approaches zero when evaluated for uniformly spaced points.

4.2 Description of Hardware and Algorithm Settings

Experiments have been performed on the Bebop computer at the Laboratory Computing Resource Center at Argonne National Laboratory. Each Broadwell node has an Intel Xeon E5-2695v4 CPU, with 36 cores and 128 GB of DDR4 RAM. A single node has been dedicated for each problem instance run and both timing and convergence data was collected. In the context of a true simulation-based MOP, this methodology models the availability of computational resources to perform up to 36 concurrent evaluations of F. In order to evaluate runtime performance, both wall time and CPU time have been measured.

Two variations of the parallel implementation from Section 3.1 are run. The first variation (`bVTdirect1`) demonstrates the true performance of Algorithm 1 using `bVTdirect` on a shared-memory system, as the total number of OpenMP threads is limited to 36. This implies that less than 36 threads can be working during each run of `bVTdirect`, since numerous threads can be waiting at any given time. In the second variation (`bVTdirect2`), the maximum number of OpenMP threads is unlimited, but the number of concurrent evaluations of F is capped at 36 using an internal counter. This simulates a distributed-memory setting where sufficient resources for 36 concurrent evaluations are available. For the `bVTdirect2` variation, one cannot accurately estimate the total CPU time since multiple instances of `bVTdirect` are allowed to timeshare on a single CPU and the function evaluation time is set by using the CPU clock. Therefore, multiple evaluations of F can occur on a single core and evaluated "concurrently" without extra CPU time.

For both variations, `bVTdirect` is given a budget of 10 iterations for the 0th search (over the entire design space) and 5 iterations in each subsequent search (over the kth LTR). For F_c, the initial trust region radius is set to 0.4 and decayed by a factor of 0.5 down to a minimum tolerance of 0.04. For `DTLZ2`, the initial trust region radius is set to 0.2 and decayed by a factor of 0.5 down to a minimum tolerance of 0.02.

For both `libEnsemble` variations, 36 workers are allocated. The 0th Latin hypercube search (over the entire design space) consists of 500 function evaluations, and all subsequent searches (over the kth LTR) consist

of 72 function evaluations. The preferred batch size for each batch of candidates is $n_b = 36$, and for the `libEnsemble2` variation, 54 random weight vectors are generated to pad out batches. The trust region radii and tolerances are the same as with `bVTdirect`.

4.3 Results

The performance statistics in this section are gathered by averaging over five repeated trials.

4.3.1 Approximation Results

Tables 1, 2, and 3 show the average number of solution points, RMSE, and Delaunay discrepancy, respectively, for the unmodified MOA (using `bVTdirect`) and the two `libEnsemble` variations. Statistics are shown for budgets of 1,000, 1,500, and 2,000 in a comma separated list. The average number of solution points is rounded to the nearest integer, and RMSE and discrepancies are rounded to three significant figures. Because the changes between `bVTdirect1` and `bVTdirect2` affect only the runtime and not the solution set, there is no need to differentiate between these two variations in this section. Similarly, there is no need to differentiate between the runs that are performed with and without runtime variance. The Delaunay discrepancy is computed by using `SciPy`, whose Delaunay triangulation algorithm can produce "flat" simplices when given grid-aligned data with $p \geq 4$. Since `bVTdirect` samples on an implicit mesh, the Delaunay discrepancy could not be evaluated for `bVTdirect` for $p = 4$. These entries are labeled "NA."

Table 1: Number of solutions found by `bVTdir`, `libE1`, and `libE2` for test problems F_c and `DTLZ2`.

Problem/Method	$p = 2$	$p = 3$	$p = 4$
F_c / `bVTdir`	42, 55, 73	81, 141, 173	103, 195, 288
F_c / `libE1`	16, 26, 38	40, 64, 93	59, 111, 171
F_c / `libE2`	48, 65, 78	67, 135, 189	105, 201, 283
`DTLZ2` / `bVTdir`	70, 108, 139	190, 286, 354	328, 481, 658
`DTLZ2` / `libE1`	48, 64, 80	102, 170, 264	243, 366, 510
`DTLZ2` / `libE2`	27, 45, 66	89, 170, 258	228, 379, 548

Table 2: RMSE reported by `bVTdir`, `libE1`, and `libE2` for test problems F_c and `DTLZ2`.

Problem/Method	$p = 2$	$p = 3$	$p = 4$
F_c / `bVTdir`	.00267, .00139, .00100	.0587, .0515, .0505	.122, .107, .101
F_c / `libE1`	.0254, .0153, .0115	.0902, .0725, .0665	.146, .127, .117
F_c / `libE2`	.0187, .0143, .0127	.0675, .0599, .0560	.123, .110, .104
`DTLZ2` / `bVTdir`	.00903, .00805, .00713	.0263, .0338, .0401	.0289, .0432, .0443
`DTLZ2` / `libE1`	.144, .112, .0993	.283, .200, .167	.289, .236, .210
`DTLZ2` / `libE2`	.163, .126, .103	.262, .192, .175	.255, .211, .201

For both problems, the `bVTdirect` variation appears to outperform both `libEnsemble` variations at a fixed function evaluation budget. For the harder problem `DTLZ2`, the difference in performance is more pronounced. This is not surprising since `bVTdirect` utilizes function evaluation information to improve performance during each RSM search phase, at the expense of parallel efficiency. Between the two `libEnsemble` variations, there is no clear best algorithm for any of the criteria.

Table 3: Discrepancy for `bVTdir`, `libE1`, and `libE2` for test problems F_c and `DTLZ2`.

Problem/Method	$p = 2$	$p = 3$	$p = 4$
F_c / `bVTdir`	.0663, .117, .207	.627, .564, .579	NA, NA, NA
F_c / `libE1`	.177, .201, .180	.513, .486, .512	.733, .676, .689
F_c / `libE2`	.181, .209, .158	.432, .560, .429	.556, .554, .551
`DTLZ2` / `bVTdir`	.132, .137, .109	.348, .221, .230	NA, NA, NA
`DTLZ2` / `libE1`	.111, .106, .139	.340, .430, .458	.635, .672, .757
`DTLZ2` / `libE2`	.218, .208, .201	.322, .528, .691	.580, .804, .793

4.3.2 Runtime Performance Results

Table 4 shows the average wall times and CPU times in seconds for solving both F_c and `DTLZ2` with and without performance variance with a budget of 2,000 function evaluations. Summary statistics are reported for $p = 2$, $p = 3$, and $p = 4$ objectives, using the `bVTdirect1`, `bVTdirect2`, `libEnsemble1` and `libEnsemble2` methods. Note that the total amount of CPU time required to perform all 2,000 evaluations is (approximately) 2,000 seconds for all problems, without considering iteration tasks. Also, recall that the CPU times could not be accurately computed for the `bVTdirect2` variation; these entries are labeled "NA."

Table 4: Runtime performance summary (CPU time / wall time).

	Method	F_c, no variance	F_c, w/ variance	`DTLZ2`, no variance	`DTLZ2`, w/ variance
$p = 2$	`bVTdir1`	2008.15 / 1037.60	2007.22 / 1039.40	2007.74 / 1093.20	2004.87 / 1082.22
	`bVTdir2`	NA / 170.58	NA / 239.09	NA / 175.28	NA / 240.15
	`libE1`	2015.46 / 88.64	2016.78 / 107.32	2027.76 / 89.38	2011.80 / 109.09
	`libE2`	2051.96 / 112.32	2070.76 / 142.66	2060.57 / 111.98	2064.93 / 143.50
$p = 3$	`bVTdir1`	2012.50 / 717.86	2012.79 / 719.34	2021.66 / 797.24	2018.79 / 797.70
	`bVTdir2`	NA / 137.08	NA / 207.48	NA / 165.54	NA / 237.07
	`libE1`	2023.13 / 94.05	2033.94 / 116.28	2039.76 / 95.31	2023.30 / 116.91
	`libE2`	2077.04 / 133.10	2066.47 / 144.46	2054.04 / 99.34	2057.03 / 126.77
$p = 4$	`bVTdir1`	2026.62 / 582.50	2029.23 / 586.49	2177.44 / 807.59	2149.43 / 782.86
	`bVTdir2`	NA / 134.42	NA / 208.84	NA / 280.75	NA / 348.38
	`libE1`	2041.90 / 107.90	2044.53 / 127.45	2176.07 / 199.70	2200.73 / 280.49
	`libE2`	2134.58 / 190.23	2124.67 / 186.57	2182.78 / 227.51	2185.34 / 257.72

For all the problems, the `libEnsemble` variations appears to perform better in terms of CPU time over wall time. However, the CPU time required by the `libEnsemble2` variation is greater than that required by the `bVTdirect1` variation because of the additional computations required for padding out the batches of candidate points. The `libEnsemble1` variation also requires slightly more CPU time than `bVTdirect1`, but it is not as pronounced. None of the implementations appear to make full utilization of all 36 cores.

In the case of `bVTdirect1`, this lack of full utilization is caused by a lack of opportunity for increased levels of concurrency. For the `libEnsemble` variations, the lack of full CPU utilization is shown in Figure 1. Figure 1 shows how the CPU is utilized over time in the three-objective case for all four problem variations with both variations of `libEnsemble`. Note that the performance peaks occur when evaluating a batch of simulations in parallel, and the performance valleys represent iteration tasks, which are not parallelized.

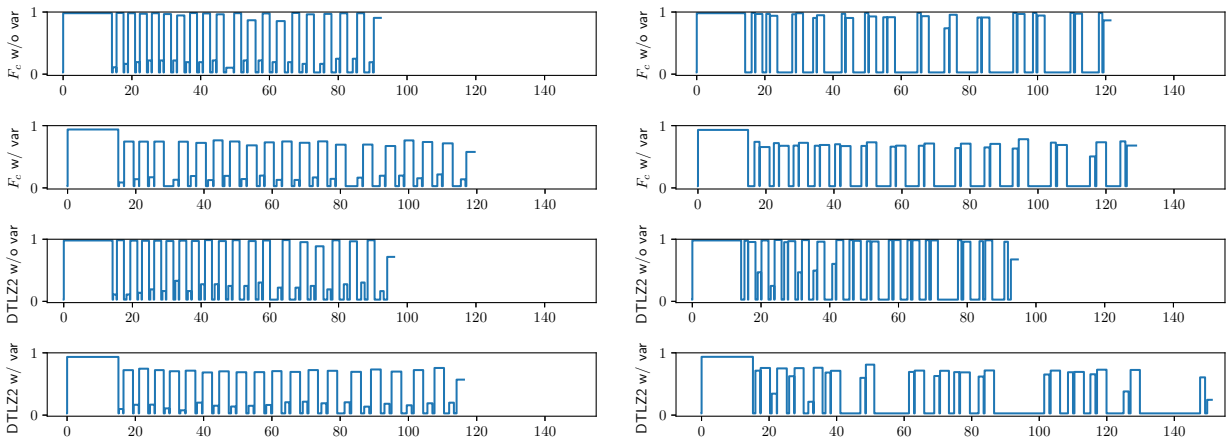

Figure 1: CPU utilization over time for a single run of `libEnsemble1` (left) and `libEnsemble2` (right) with $p = 3$ objectives on both test problems, with and without runtime variance. The y-axis shows proportion of CPU resources being utilized and the x-axis shows the time in seconds since the beginning of the computation.

In the usage plot for `libEnsemble1`, half the performance peaks approach nearly 100% CPU utilization, while the other half utilize only a small fraction of the 36 available cores. This is expected since the `libEnsemble1` variation makes no effort to pad out each batch of candidate solutions. In the case of `libEnsemble2`, all of the performance peaks approach 100% CPU utilization for the convex problem F_c. For the nonconvex problem `DTLZ2`, it is impossible to find n_a unique solutions to fully pad out each batch of candidate designs. The overall CPU utilization for the `libEnsemble2` variation is consistently worse, due to the width of the performance valley for each iteration task. These iteration tasks take longer for the `libEnsemble2` variation because of the added expense of solving additional surrogate optimization problems to pad out the candidate design batches. For real-world objective functions involving expensive simulations, the evaluation times are often significantly greater than one second and would overwhelm these heightened iteration costs. Therefore, the `libEnsemble2` variation is recommended.

5 DISCUSSION

In this paper, several variations of a recent MOA are implemented and demonstrated on several analytic problems. One of these implementations leverages a variation of `VTDIRECT95`, and the other two integrate with `libEnsemble` and are tailored for load balancing. The "`libEnsemble2`" variation is recommended in an HPC setting since it would achieve better HPC resource utilization for computationally expensive cost functions. However, the "`bVTdirect`" implementation can achieve slightly better approximations to the Pareto front for a fixed function evaluation budget.

ACKNOWLEDGMENTS

This work was supported by the U.S. Department of Energy (DOE) through the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations, the Office of Science and the National Nuclear Security Administration. This work was also supported by the DOE Office of Science Graduate Student Research (SCGSR) program. The SCGSR program is administered by the Oak Ridge Institute for Science and Education (ORISE), which is managed by ORAU under contract number DE-SC0014664. All opinions in this paper are the authors' and do not necessarily reflect the policies and views of DOE, ORAU, or ORISE. The authors gratefully acknowledge the HPC computing resources provided on Bebop, operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

REFERENCES

Audet, C., J. Bigeon, D. Cartier, S. Le Digabel, and L. Salomon. 2018. "Performance indicators for multiobjective optimization", *Optimization Online* Article 2018/10/6887, 39 pages.

Campana, E., M. Diez, L. Matteo, G. Liuzzi, S. Lucidi, R. Pellegrini, V. Piccialli, F. Rinaldi, and A. Serani. 2018. "A multi-objective DIRECT algorithm for ship hull optimization", *Comp. Optim. and App.* vol. 71(1), pp. 53–72.

Chang, T. H., L. T. Watson, T. C. H. Lux, B. Li, L. Xu, A. R. Butt, K. W. Cameron, and Y. Hong. 2018. "A polynomial time algorithm for multivariate interpolation in arbitrary dimension via the Delaunay triangulation". In *Proc. ACMSE 2018 Conference* Article No. 12.

Deb, K., A. Pratap, S. Agarwel, and T. Meyarivan. 2002a. "A fast and elitist multiobjective genetic algorithm: NSGA-II", *IEEE Trans. Evolutionary Computation* vol. 6(2), pp. 182–197.

Deb, K., L. Thiele, M. Laumanns, and E. Zitzler. 2002b. "Scalable multi-objective optimization test problems". In *Proc. 2002 IEEE Congress on Evolutionary Computation* vol. 1, pp. 825–830.

Deshpande, S., L. T. Watson, and R. A. Canfield. 2016. "Multiobjective optimization using an adaptive weighting scheme", *Optimization Methods and Software* vol. 31(1), pp. 110–133.

Ehrgott, M. 2005. *Multicriteria Optimization*. Heidelberg, Germany, Springer Science & Business Media.

He, J., L. T. Watson, and M. Sosonkina. 2009. "Algorithm 897: VTDIRECT95: Serial and parallel codes for the global optimization algorithm DIRECT", *ACM Trans. Math. Softw.* vol. 36(3), Article No. 17.

Hudson, S., J., Larson, S. M. Wild, D. Bindel, and J.-L. Navarro. 2019. "libEnsemble users manual". Revision 0.6.0, Argonne National Laboratory, Lemont, IL.

Kodiyalam, S., R. J. Yang, L. Gu, and C.-H. Tho. 2004. "Multidisciplinary design optimization of a vehicle system in a scalable, high performance computing environment", *Structural and Multidisciplinary Optim.* vol. 26(3), pp. 256–263.

Kugele, S. C., M. W. Trosset, and L. T. Watson. 2008. "Numerical integration in statistical decision-theoretic methods for robust design optimization", *Structural and Multidisciplinary Optim.* vol. 36(5), pp. 457–475.

Le Digabel, S. 2011. "Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm", *ACM Trans. Math. Softw.* vol. 37(4), Article No. 44.

Myers, R. H., D. C. Montgomery, and C. M. Anderson-Cook. 2016. *Response Surface Methodology: Process and Design Optimization Using Designed Experiments*. John Wiley & Sons, Inc.

Thacker, W. I., J. Zhang, L. T. Watson, J. B. Birch, M. A. Iyer, and M. W. Berry. 2010. "Algorithm 905: SHEPPACK: Modified Shepard algorithm for interpolation of scattered multivariate data", *ACM Trans. Math. Softw.* vol. 37(3), Article No. 34.

AUTHOR BIOGRAPHIES

TYLER H. CHANG is a Ph.D. candidate and Cunningham fellow at Virginia Tech, advised by Layne Watson. He is interested in numerical analysis, algorithms, and parallel computing. His email is thchang@vt.edu.

JEFFREY LARSON (Ph.D., University of Colorado Denver, 2012) is a computational mathematician at Argonne National Laboratory and a lead libEnsemble developer. He studies algorithms for optimizing computationally expensive functions. His email is jmlarson@anl.gov.

LAYNE T. WATSON (Ph.D., Michigan, 1974) is a professor at Virginia Tech. He has interests in numerical analysis, mathematical programming, bioinformatics, and data science. He has helped with the organization of HPCS since 2000. His email is ltw@cs.vt.edu.

THOMAS C. H. LUX is a Ph.D. candidate at Virginia Tech, advised by Layne Watson. His interests include approximation theory, optimization, and artificial intelligence. His email is tchlux@vt.edu.