Scaling Exact Multi-Objective Combinatorial Optimization by Parallelization

Jianmei Guo, Edward Zulkoski, Rafael Olaechea, Derek Rayside, Krzysztof Czarnecki
University of Waterloo, Canada

Sven Apel
University of Passau, Germany

Joanne M. Atlee
University of Waterloo, Canada

ABSTRACT

Multi-Objective Combinatorial Optimization (MOCO) is fundamental to the development and optimization of software systems. We propose five novel parallel algorithms for solving MOCO problems exactly and efficiently. Our algorithms rely on off-the-shelf solvers to search for exact Pareto-optimal solutions, and they parallelize the search via collaborative communication, divide-and-conquer, or both. We demonstrate the feasibility and performance of our algorithms by experiments on three case studies of software-system designs. A key finding is that one algorithm, which we call FS-GIA, achieves substantial (even super-linear) speedups that scale well up to 64 cores. Furthermore, we analyze the performance bottlenecks and opportunities of our parallel algorithms, which facilitates further research on exact, parallel MOCO.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*software configuration management*; G.1.6 [**Numerical Analysis**]: Optimization—*constrained optimization*

Keywords

Multi-objective combinatorial optimization, parallelization

1. INTRODUCTION

Multi-Objective Combinatorial Optimization (MOCO) explores a finite search space of feasible solutions and finds the optimal ones that balance multiple (often conflicting) objectives simultaneously. MOCO is a fundamental challenge in many design and development problems in engineering and other domains. For example, in mobile-phone system design, one often has to choose between different candidate designs that trade off multiple competing *objectives*, such as low cost and high performance. Each candidate design (i.e., a feasible solution) involves a wide variety of design options, which we call *features* (e.g., enabling Video Calls) [11], with respect to a set of *constraints* (e.g., Video Calls requires Camera) and *quality attributes* (e.g., the cost of enabling Video Calls). In the worst case, the search space of candidate designs grows exponentially in the number of features. Exploring such a huge search space is often beyond human capabilities and makes optimal system design a very challenging task. In the same way, many software-engineering problems, such as architecture design [1], test data generation [23, 44], and project planning [20], involve the same form of MOCO.

MOCO problems are mostly NP-hard [10]. To address them, approximate approaches that depend mainly on *metaheuristics*[1] have been advocated for years. In most cases, they solve MOCO problems in an acceptable time, but they find only near-optimal solutions, and often suffer from parameter sensitivity (i.e., the accuracy of the found solutions varies widely with the parameter settings of these approaches) [18]. In contrast, exact methods that scan all candidate solutions one by one often take too long for large-scale problems, but they are accurate in finding *all*, *exact* optimal solutions, which is desirable for those stakeholders who never want to miss any optimal opportunity.

Parallel computing carries out multiple calculations simultaneously on multiple processors [2]. It divides a large computing problem into multiple smaller ones and solves them in parallel, often with a significant performance improvement. In the past, metaheuristics have been parallelized to address MOCO problems efficiently [7, 39]. However, there are only few parallel algorithms for exact MOCO [39].

We aim at exact, parallel approaches that solve MOCO problems accurately and efficiently. As a baseline, we choose the *Guided Improvement Algorithm (GIA)* [33], a general-purpose, sequential algorithm for solving MOCO problems exactly. GIA works with most off-the-shelf SAT (SATisfiability), SMT (Satisfiability Modulo Theories) [12], and CSP (Constraint Satisfaction Problem) [41] solvers. Accordingly, the first parallel algorithm we propose, which we call *Parallel GIA (ParGIA)*, performs multiple GIAs simultaneously and collaboratively. In ParGIA, each processor runs a GIA with a different starting point in the search space; once a processor finds an optimal solution, it communicates the solution to other processors so as to reduce duplicate searches.

[1] A metaheuristic is a strategy for exploring the search space of a problem using a variety of methods that typically consist of both a diversification (i.e., mechanisms to explore the search space) and an intensification (i.e., mechanisms that exploit previously found solutions) procedure [6].

To further scale MOCO, we propose two parallel divide-and-conquer algorithms, *Objective Split GIA (OS-GIA)* and *Feature Split GIA (FS-GIA)*. OS-GIA geometrically divides the search space of a MOCO problem into subspaces, and then runs in parallel a GIA for each subspace. FS-GIA recursively partitions a MOCO problem into subproblems by selecting and deselecting certain features; then it performs in parallel a GIA for each subproblem. Lastly, we propose two hybrid parallel algorithms, *OS-ParGIA* and *FS-ParGIA*, in which the GIA algorithm in OS-GIA and FS-GIA, respectively, is replaced with ParGIA.

We implemented our proposed algorithms and evaluated them in a series of experiments on three case studies of software-system design. Our empirical results demonstrate the feasibility and performance of our parallel algorithms. In particular, FS-GIA shows a desirable scalability with an increasing number of available processors, and it achieves even super-linear speedups.[2]

In summary, we make the following contributions:

- **Algorithms.** We propose five novel parallel MOCO algorithms that search for exact optimal solutions using off-the-shelf solvers, and that parallelize the search via collaborative communication, divide-and-conquer, or both.
- **Implementation.** We implement our proposed algorithms and publish the source code and experimental data at `http://gsd.uwaterloo.ca/epoal`.
- **Evaluation.** We evaluate the performance and scalability of our parallel algorithms and analyze their performance bottlenecks.
- **Prospects.** We open a new direction in scaling exact MOCO algorithms and demonstrate the potential of parallelization for exact MOCO.

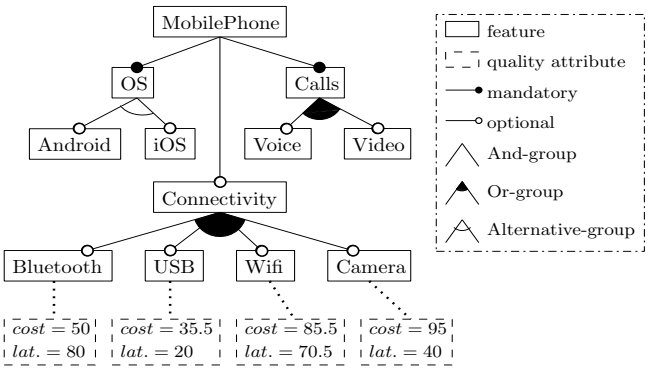
2. PRELIMINARIES

First, we introduce MOCO by means of the example of mobile-phone system design, including a set of features, constraints, quality attributes, and objectives. Next, we describe how to specify features and their constraints in propositional formulas and how to use solvers to acquire feasible solutions. Finally, we describe the concepts of Pareto-optimal solutions and present how GIA (our baseline, sequential algorithm) finds all Pareto-optimal solutions and, this way, solves MOCO problems exactly.

2.1 A Running Example

The design of a software system deployed on a mobile phone can be characterized by the *features* supported by the phone and the *constraints* defined between the features. The features represent various design options. For example, the phone's software may optionally include support for feature Video Calls. Figure 1 shows a design scenario of a mobile-phone system, adapted from Benavides et al. [5]. We organize and visualize its features and their constraints using a *feature model* [26], a tree structure, in which each node except the root (i.e., MobilePhone) has one parent. Relations between a feature and its group of child features are

[2]Speedup is defined as $S_P = \frac{T_1}{T_P}$, where T_1 is the sequential execution time of a problem and T_P is the parallel execution time of the same problem using P processors. Linear speedup is obtained when $S_P = P$ and super-linear speedup is achieved when $S_P > P$.



Cross-tree constraints : *Video requires Camera*

Objectives : *minimizing cost, minimizing latency*

Figure 1: A sample MOCO problem arising in mobile-phone system design

classified as And- (no arc), Or- (filled arc), and Alternative-groups (arc). The members of an And-group can be either mandatory (filled circle) or optional (empty circle). Cross-tree constraints comprise requires and excludes relations between features (e.g., Video requires Camera).

Stakeholders can customize the mobile-phone system by selecting features, thereby deriving different design variants that have different *quality attributes* (e.g., cost and latency). Each feature may have an influence on the quality attributes of a feasible system design that can be implemented and measured [17]. For example, selecting (or deselecting) feature Bluetooth in Figure 1, increases (or reduces) the cost by 50 and the latency by 80 in the final mobile-phone system. The quality attributes of a feasible system design can be calculated by aggregating the quality attributes of all selected features and feature interactions involved [38]. In this paper, we assume that we already have the quality attributes of features and potential feature interactions. For research on measuring and inferring the quality attributes, we refer the interested reader elsewhere [16, 38, 42].

Among the many feasible design variants, stakeholders often desire the optimal one that can simultaneously meet multiple design *objectives*, such as minimizing cost and minimizing latency. This is a typical MOCO problem. However, the objectives are often conflicting. For example, a mobile-phone system often achieves a lower latency only by raising its cost (e.g., using a larger cache). Thus, a MOCO problem usually has a set of optimal solutions, not only a single one. Finding all optimal solutions accurately and efficiently is a major challenge, which we address in this paper.

2.2 Specification and Validation

The solutions to a MOCO problem must be *feasible*, i.e., satisfying the constraints defined between features. To this end, we specify all features and their constraints in propositional formulas, and then we use off-the-shelf SAT, SMT, or CSP solvers to return feasible solutions. Note that combinatorial optimization, either single-objective or multi-objective, explores a *finite* search space, and each feature ranges over a finite domain [10]. In general, a finite-domain feature can be converted to a finite set of Boolean-domain features [8]. Here, we use only Boolean-domain features to describe MOCO problems. In a nutshell, we represent a feature as

Table 1: Feature-selection constraints in propositional logic (P represents a parent feature and $C_1, ..., C_n$ are its child features; $M \subseteq \{1, ..., n\}$ denotes the mandatory features by their indices in an And-group; F_1 and F_2 denote arbitrary features)

Type	Propositional Formulas
Mandatory	$C_1 \leftrightarrow P$
Optional	$C_1 \rightarrow P$
And-group	$(P \rightarrow \bigwedge_{i \in M} C_i) \wedge (\bigvee_{1 \leq i \leq n} C_i \rightarrow P)$
Or-group	$P \leftrightarrow \bigvee_{1 \leq i \leq n} C_i$
Alternative-group	$(P \leftrightarrow \bigvee_{1 \leq i \leq n} C_i) \wedge \bigwedge_{i<j} (\neg C_i \vee \neg C_j)$
Requires	$F_1 \rightarrow F_2$
Excludes	$\neg(F_1 \wedge F_2)$

a Boolean decision variable. If a feature is supported by a system design, then its corresponding variable is assigned `true`, and `false` otherwise. Furthermore, we specify the constraints between features in propositional formulas. For example, the constraints defined in feature models can be formulated in propositional logic, as summarized in Table 1 [3].

A *solution* is a selection of features, i.e., an assignment of value `true` or `false` to the decision variable of each feature. A feasible solution is a *valid* selection of features that respects all constraints defined between features. Using off-the-shelf solvers, we are able to check whether a potential solution satisfies all constraints, and thus is a feasible solution [3]. For example, by using a SAT solver, we determine that solution {OS, Android, Calls, Video, Connectivity, Camera} in Figure 1 is feasible, but solution {OS, Android, Calls, Video, Connectivity, Wifi} is not, because the latter violates the constraint: Video requires Camera.

2.3 Pareto-Optimal Solutions and GIA

As a baseline, we use GIA (Guided Improvement Algorithm) [33] to determine whether a feasible solution is Pareto-optimal and where to direct the search next. Figure 2 illustrates the *search space* of the MOCO problem of Figure 1, in which each point indicates a solution.[3] Given multiple potentially conflicting objectives, a solution is *Pareto-optimal* if it is not dominated by any other solution. A solution *dominates* another solution when it is better regarding at least one objective and not worse regarding all the other objectives. According to the definition of Pareto dominance, a solution partitions a search space into three areas: *inferior*, *superior*, and *equilibrium*. For example, in Figure 2, the inferior area (top-right) of solution S_5 includes all solutions that are dominated by S_5. The superior area (bottom-left) contains all solutions that dominate solution S_5. Any solution in the equilibrium areas (top-left and bottom-right) does not dominate solution S_5 and, at the same time, is not dominated by S_5. All Pareto-optimal solutions (filled points) constitute the *Pareto front*.

GIA uses a solver to return a solution and then augments the constraints to search for solutions that dominate ones found already. Moreover, GIA incrementally finds Pareto-optimal solutions during computation and thus guarantees that all solutions yielded by the algorithm are Pareto-optimal, even if one terminates the execution halfway. This gains a competitive advantage over other exact methods [15, 29].

[3]Unless otherwise specified, all solutions mentioned in the following sections are feasible.

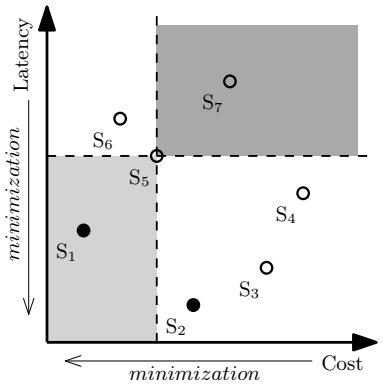


Figure 2: Solution S_5 partitions the search space into inferior (top-right), superior (bottom-left), and equilibrium (top-left and bottom-right) areas

Algorithm 1: Guided Improvement Algorithm (GIA)

input : M, A, O
output: S

```
1  S ← ∅
2  EqC ← true
3  s ← solveOne(M)
4  while s ≠ null do
5      while s ≠ null do
6          s' ← s
7          SupC ← genSupC(s, M, A, O)
8          s ← solveOne(M ∧ EqC ∧ SupC)
9      S ← S ∪ {s'}
10     EqC ← EqC ∧ genEqC(s', M, A, O)
11     s ← solveOne(M ∧ EqC)
12 return S
```

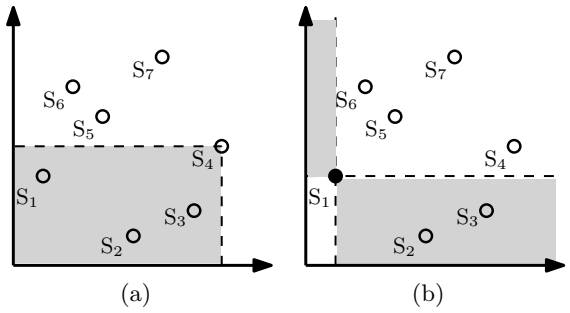


Figure 3: (a) GIA returns solution S_4 and calculates the constraints of the superior area of S_4; (b) GIA finds a Pareto-optimal solution S_1 and calculates the constraints of the equilibrium area of S_1

Algorithm 1 lists the pseudo-code of GIA. GIA starts with a model M, a set of quality attributes A, and multiple objectives O. The model M specifies all features and their constraints as propositional formulas. The output is the Pareto front S. As demonstrated in Figure 3a, GIA first inputs a

model into a solver and computes a solution S_4 (Line 3 in Algorithm 1). GIA calculates the constraints of the superior area of solution S_4 (grey area in Figure 3a) and adds them to the solver for finding another solution dominating S_4 (Lines 7–8). A Pareto-optimal solution has been found when there is no solution in its superior area. As shown in Figure 3b, solution S_1 is marked as Pareto-optimal (filled). Next, GIA calculates the constraints of the equilibrium area of solution S_1 (grey areas in Figure 3b) and adds them to the solver for finding other Pareto-optimal solutions (Lines 10–11).

3. PARALLEL ALGORITHMS

In this section, we present five novel parallel algorithms for exactly solving MOCO problems. We use the example of Figures 1 and 2 to explain them. The input of each algorithm includes a model M specifying features and constraints, a set of quality attributes A, multiple objectives O, and the number of available processors P. The output is the Pareto front S.

3.1 ParGIA

ParGIA uses P processors to perform P GIAs simultaneously and collaboratively over the search space of a given MOCO problem. Each processor runs a GIA instance to search for Pareto-optimal solutions and then communicates the constraints of the found solutions to all other processors. Algorithm 2 lists the pseudo-code of ParGIA. Figure 4 illustrates the search process of ParGIA using two processors. Processor P_1 inputs a model into a solver and computes a solution S_7; P_1 keeps searching until it finds a Pareto-optimal solution S_1. Meanwhile, processor P_2 starts with a solution S_4 and then finds another Pareto-optimal solution S_2. If processors P_1 and P_2 find S_1 and S_2 at exactly the same time, they communicate to each other the constraints of the equilibrium areas of S_1 and S_2 (Lines 11–12 in Algorithm 2). Next, both processors search the *combined* equilibrium area (grey areas in Figure 4b), looking for other Pareto-optimal solutions. However, in most cases, the search processes of processors P_1 and P_2 are not synchronous. For example, processor P_1 may find solution S_1 before processor P_2 reaches solution S_2. In this case, processor P_1 communicates the constraints of the equilibrium area of S_1 to processor P_2, but it does not receive any constraints from P_2. Subsequently, processor P_1 keeps searching for other Pareto-optimal solutions in the equilibrium area of S_1, which may cause the *overlapping* search of solution S_2 that will be found by processor P_2 later. Following the idea of *optimistic parallelism* [27], ParGIA does not control or avoid potentially overlapping search of the same Pareto-optimal solutions. But, after all processors finish searching, ParGIA checks all found Pareto-optimal solutions and removes duplicates (Line 14).

3.2 OS-GIA and OS-ParGIA

OS-GIA geometrically divides the search space of a given MOCO problem into P subspaces and then simultaneously conquers all subspaces using P processors. In each subspace, a GIA instance is performed by one processor to search for Pareto-optimal solutions. Algorithm 3 lists the pseudo-code of OS-GIA. We adopt the idea of *cone separation* [7] to divide the search space. First, OS-GIA converts all objectives to be all minimizations or all maximizations (Line 2 in Algorithm 3). Note that an objective to maximize X is equivalent to the objective to minimize $-X$. Without loss of generality, we assume that all objectives involve minimizations, as the example of Figures 1 and 2. Then, OS-GIA determines a reference point (the crossed point in Figure 5a) whose value regarding each objective is sufficiently large, such that all solutions are located in its superior area.[4]

If the given MOCO problem has more than two objectives, OS-GIA projects the search space onto a bi-dimensional plane formed by two chosen objectives (Line 3). Next, OS-GIA divides the projective plane into *cones* by, starting from the reference point, dividing the 90° angle encompassing the superior area of the reference point into P equal parts (Lines 4–5). Figure 5a shows the result of dividing the projective into two cones; each cone has a 45° angle starting from the reference point. Finally, OS-GIA applies a GIA to each cone to search for Pareto-optimal solutions (Line 7).

The overhead of geometric decomposition is very small. However, choosing the objectives that determine the projective plane and cones plays a critical role in the eventual

[4]If all objectives are maximizations, the reference must be small enough regarding all the objectives.

Algorithm 2: ParGIA

input : M, A, O, P
output: S

```
1  S ← ∅
2  forall i ← 1 to P in parallel do
3      EqC_i ← true
4      s ← solveOne(M)
5      while s ≠ null do
6          while s ≠ null do
7              s' ← s
8              SupC_i ← genSupC(s, M, A, O)
9              s ← solveOne(M ∧ EqC_i ∧ SupC_i)
10         S ← S ∪ {s'}
11         for j ← 1 to P do
12             EqC_j ← EqC_j ∧ genEqC(s', M, A, O)
13         s ← solveOne(M ∧ EqC_i)
14 S ← postprocess(S)
15 return S
```

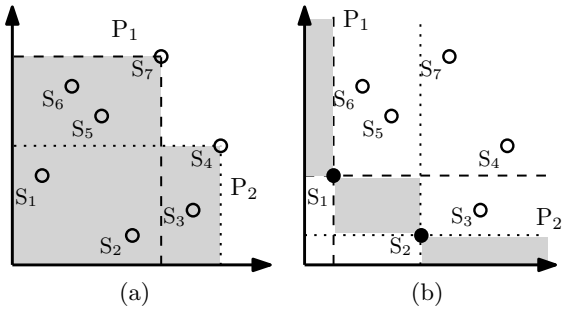


Figure 4: (a) ParGIA finds two solutions S_7 and S_4 simultaneously using two processors and calculates the constraints of the superior areas of the two solutions; (b) ParGIA finds two Pareto-optimal solutions S_1 and S_2 and calculates the constraints of the combined equilibrium areas of the two solutions

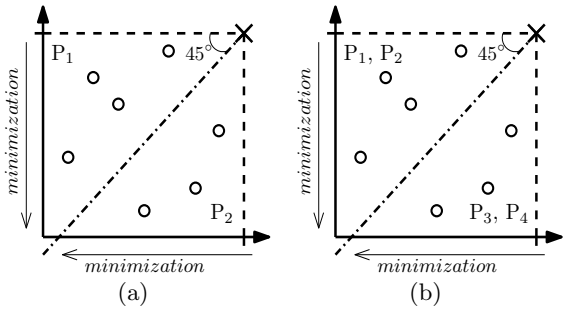


Figure 5: After geometrically dividing the search space of Figure 2 into two cones, (a) OS-GIA performs a GIA instance using one processor in each cone, and (b) OS-ParGIA performs a ParGIA instance using two processors in each cone

Algorithm 3: OS-GIA

input : M, A, O, P
output: S

1 $S \leftarrow \emptyset$
2 $O \leftarrow preprocess(O)$
3 $O_2 \leftarrow chooseTwo(O)$
4 **for** $j \leftarrow 1$ **to** P **do**
5 $\quad M_j \leftarrow objectiveSplit(M, A, O_2, P, j)$
6 **forall** $j \leftarrow 1$ **to** P **in parallel do**
7 $\quad S_j = GIA(M_j, A, O)$
8 $\quad S \leftarrow S \cup S_j$
9 $S \leftarrow postprocess(S)$
10 **return** S

Algorithm 4: OS-ParGIA

input : M, A, O, P
output: S

1 $S \leftarrow \emptyset$
2 $Q \leftarrow P/2$
3 $O \leftarrow preprocess(O)$
4 $O_2 \leftarrow chooseTwo(O)$
5 **for** $j \leftarrow 1$ **to** Q **do**
6 $\quad M_j \leftarrow objectiveSplit(M, A, O_2, Q, j)$
7 **forall** $j \leftarrow 1$ **to** Q **in parallel do**
8 $\quad S_j = ParGIA(M_j, A, O, 2)$
9 $\quad S \leftarrow S \cup S_j$
10 $S \leftarrow postprocess(S)$
11 **return** S

workload of each processor for each cone. In the worst case, a poor selection of the two objectives may cause all solutions to fall into one cone of the plane, which results in one processor overloaded and the others idle. To balance the workload of different processors, we follow a straightforward intuition of choosing the two objectives that have the first and second largest value ranges to form a large projective plane, in which all solutions are distributed as evenly as possible.

Note that the Pareto-optimal solutions found by each processor are *local* to that processor's cone. Moreover, a solution that is locally Pareto-optimal in a cone may not be globally Pareto-optimal in the entire search space. Therefore, after all processors finish searching, OS-GIA collects all local Pareto fronts found by each processor (Line 8), removes duplicate solutions, and calculates the *global* Pareto front in the search space (Line 9). Such calculation is straightforward and usually takes little time.

OS-ParGIA is a hybrid of OS-GIA and ParGIA. Algorithm 4 lists the pseudo-code of OS-ParGIA. As shown in Figure 5b, the key difference between OS-ParGIA and OS-GIA is that OS-ParGIA searches each cone using a ParGIA instance with two processors instead of using a GIA instance with one processor (Line 8 in Algorithm 4). Moreover, the input P of OS-ParGIA is double the number of divided cones (Line 2). Of course, one can perform ParGIA using more than two processors per cone, but the performance gains are not necessarily higher when using more processors, as we demonstrate in Section 5.3.

3.3 FS-GIA and FS-ParGIA

FS-GIA divides a given MOCO problem into P subproblems and then simultaneously conquers all subproblems using P processors. For each subproblem, a GIA instance is performed by one processor to search for Pareto-optimal solutions. The key idea of FS-GIA is to divide a MOCO problem into subproblems of relatively *equal size*, where each subproblem is defined by a partial feature selection that represents a subset of solutions. If a MOCO problem has N solutions in total, FS-GIA partitions the set of N solutions into P subsets of roughly equal size (ideally, $\frac{N}{P}$ solutions). Subsequently, FS-GIA uses the GIA algorithm to search for Pareto-optimal solutions among each subset. The approach has the potential to significantly reduce the number of solutions that each processor has to explore (by up to a factor of P), leading to a significant performance improvement. Algorithm 5 lists the pseudo-code of FS-GIA.

A major challenge of FS-GIA is to accurately and quickly calculate the number of all solutions selecting or deselecting a certain feature. This reduces to the well-known *#SAT problem* [40], where the goal is to count the number of satisfying variable assignments $\#(x)$ for a given propositional formula x. Although this is generally a hard problem, existing #SAT solvers, such as SHARPSAT, which we use for FS-GIA,[5] are capable of quickly calculating the exact number of solutions for the propositional formulas of our case studies, as we demonstrate in Section 5.4.

To acquire subproblems as equal-sized as possible, FS-GIA *recursively* chooses an appropriate feature and partitions each MOCO subproblem into a subsubproblem that includes the feature and a subsubproblem that does not (Lines 2–4 in Algorithm 5). As shown in Figure 6, FS-GIA starts with the input model M of a given MOCO problem. FS-GIA exhaustively tests every feature f and computes the number of solutions for submodels $M \wedge f$ and $M \wedge \neg f$ (i.e., the number of valid assignments when the corresponding variable of f is `true` or `false`). Then, FS-GIA chooses the best *split* feature (e.g., f_1 in Figure 6) to divide model M into two submodels such that:

$$|\#(M \wedge f) - \#(M \wedge \neg f)| \text{ is minimal} \quad (1)$$

If multiple features satisfy the above equation, one is randomly chosen amongst them. Each submodel is divided recursively by further split features, such as features f_2 and

[5] `https://github.com/marcthurley/sharpSAT`.

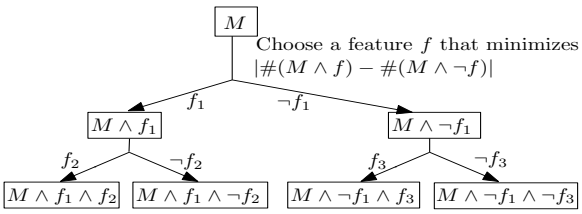


Figure 6: FS-GIA recursively chooses features f_1, f_2, and f_3 to divide a given MOCO problem into four equal-sized subproblems

Algorithm 5: FS-GIA

input : M, A, O, P
output: S

```
1 S ← ∅
2 F ← parallelPreprocess(M, P)
3 for j ← 1 to P do
4 |  M_j ← generate(M, F, j)
5 forall j ← 1 to P in parallel do
6 |  S_j = GIA(M_j, A, O)
7 |  S ← S ∪ S_j
8 S ← postprocess(S)
9 return S
```

Algorithm 6: FS-ParGIA

input : M, A, O, P
output: S

```
1  S ← ∅
2  Q ← P/2
3  F ← parallelPreprocess(M, Q)
4  for j ← 1 to Q do
5  |  M_j ← generate(M, F, j)
6  forall j ← 1 to Q in parallel do
7  |  S_j = ParGIA(M_j, A, O, 2)
8  |  S ← S ∪ S_j
9  S ← postprocess(S)
10 return S
```

f_3 in Figure 6. The above recursive division process forms a binary decision tree, as shown in Figure 6. Note that the submodels generated at the same *level* of the tree are equal-sized and non-overlapping. They represent the subproblems we need for FS-GIA. Hence, the number of generated subproblems has to be a power of two, which equals to the number of assigned processors P, since each subproblem is assigned one processor.

If we carry out the recursive division process of FS-GIA sequentially, the dividing overhead increases linearly with P. As shown in Figure 6, FS-GIA has to choose three split features and divide three times to acquire four subproblems for four processors. To reduce the dividing overhead of FS-GIA, we parallelize the preprocessing of FS-GIA (Line 2 in Algorithm 5) to choose split features simultaneously. For example, in Figure 6, the preprocessing tasks of choosing features f_2 and f_3 can be run simultaneously. In general, subproblems with the same number of split-feature clauses can be further divided in parallel: in step i, 2^{i-1} subproblems are split roughly in half, producing 2^i smaller subproblems. Thus, the overhead of partitioning the model increases *logarithmically* with P.

FS-ParGIA is a hybrid of FS-GIA and ParGIA. Algorithm 6 lists the pseudo-code of FS-ParGIA. The key difference between FS-ParGIA and FS-GIA is that, for each subproblem, FS-ParGIA performs a ParGIA using multiple processors instead of a GIA using one processor. For example, Algorithm 6 (Lines 2 and 7) addresses each subproblem using two processors. Hence, the input P of FS-ParGIA is double the P of FS-GIA, when addressing the same number of subproblems. As with OS-GIA and OS-ParGIA, the Pareto fronts found by FS-GIA and FS-ParGIA are local to the corresponding subproblems. Thus, a post-processing step (Algorithm 5, Line 8, and Algorithm 6, Line 9) identifies which locally Pareto-optimal solutions make up the global Pareto front with respect to the entire search space. This step takes proportionately little time.

4. IMPLEMENTATION

We implemented our parallel algorithms using PYTHON 2.7 and its MULTIPROCESSING package. The MULTIPROCESSING package effectively steps aside the issue of *Global Interpreter Lock* by using subprocesses instead of threads.[6] It allows programmers to fully leverage multiple processors on a given machine, in which each spawned subprocess is assigned a processor.

Following the idea of optimistic parallelism [27] and reducing the communication cost for ParGIA, we avoid using any synchronization primitives, such as locks, but we use *message-passing* mechanisms, such as queues, which provide a process-safe communication channel. Thus, each processor in ParGIA has its own queue to manage the constraints of the Pareto-optimal points found by itself, and then communicates the constraints to the queues of other processors.

A recent study [35] shows that SMT might be the most efficient reasoning formalism in checking model properties, compared to CSP, Alloy [24], and Answer Set Programming (ASP) [30]. SMT combines standard SAT with richer theories, such as equality reasoning, linear arithmetic, bitvectors, and arrays [12]. Therefore, in our implementation, we reason about MOCO problems in SMT solvers. In particular, we implemented GIA and all five parallel algorithms with the efficient SMT solver Z3, developed at Microsoft Research.[7]

5. EVALUATION

We conducted a series of experiments to evaluate the parallel algorithms we propose. We aim at investigating the performance and scalability of each algorithm and at identifying potential bottlenecks and opportunities.

5.1 Subjects

We evaluated our parallel algorithms on three MOCO case studies from the domain of software-system designs. As MOCO problems reported by industry [22, 25, 34] are not

[6]Global Interpreter Lock is the mechanism used by PYTHON interpreter to assure that only one thread executes PYTHON bytecode at a time.

[7]`http://z3.codeplex.com`.

Table 2: Overview of subject systems

	#Features	#Solutions	#Objectives
SAS	35	5 184	7
Web Portal	44	2 120 800	4
E-Shop	290	5.02E+49	4

available publicly, we resorted to using three case studies from existing literature [14, 37] as subjects. The first subject, SAS, optimizes the architecture of a real-world situation-awareness system that deploys personnel in emergency response scenarios [14]. The original authors collected seven integer and float quality attributes for each feature. Accordingly, SAS has seven objectives: minimizing cost, maximizing reliability, minimizing battery usage, minimizing response time, minimizing ramp-up time, minimizing development time, and minimizing deployment time.

Web Portal and E-Shop are two product-line design models available at the SPLOT website [31], which is a popular repository of software-system models used by many researchers. Sayyad et al. [37] extended the two models and formulated corresponding MOCO problems by adding integer and float quality attributes for each feature and defining a set of objectives. They randomly generated quality attributes resembling real-world project characteristics [21]. We use the same data and target the same objectives they defined for both case studies: minimizing cost, minimizing the number of defects, maximizing the number of offered features, and maximizing the number of features that were used before.

As listed in Table 2, these case studies cover a reasonable spectrum of MOCO problems with different characteristics: different number of features (35 to 290), different number of solutions (3 to 49 orders of magnitude), different number of objectives (4 to 7), different quality attributes (e.g., cost, response time, battery usage, reliability, and software defects), different value types of quality attributes (integer and float), and different optimization directions (minimization and maximization). Furthermore, finding the Pareto front of these case studies takes different time ranges (from seconds to minutes and days).

5.2 Experimental Setup

We conducted our experiments on SHARCNET, which is a consortium of Canadian academic institutions that share a network of high-performance computers.[8] To reduce fluctuations in measurements caused by different hardware environments, we performed all measurements on the same cluster comprising 160 AMD Opteron cores each at 2.2 GHz and a total of 640 GB RAM. However, we were not able to occupy all resources (e.g., cores and memory) on the cluster at all times, because SHARCNET uses a priority queue to schedule the jobs submitted to a cluster. The priority of a job is ranked according to the resources requested by the job. More resources requested result in lower priority and more waiting time in the queue. Due to the limitation of resources and time, our experiments use up to 64 cores simultaneously for each job submitted to the cluster.

In our experiments, the *independent variables* are the subject systems, the evaluated algorithms, and the number of assigned cores. We measured the execution time of each sequential and parallel algorithm finding the Pareto front of a certain subject system, and we calculated the speedups of each parallel algorithm as the *dependent variables*. Unfortunately, we were not able to determine all Pareto-optimal solutions of E-Shop using any of the algorithms in six days, which is the maximum execution time allocated to a job submitted to SHARCNET. Hence, we measured the number of Pareto-optimal solutions found by each algorithm in six days as the dependent variable for E-Shop.

To reduce fluctuations in the values of dependent variables caused by randomness (e.g., the random seeds used by solvers to return a solution), we evaluated each combination of the independent variables 10 times. That is, for each subject system, we executed each of the algorithms with the same number of assigned cores 10 times, and we measured the resulting execution times. We report only the means of the execution times for analysis.

As a baseline, we performed sequential GIA on the three subjects for performance comparison and speedup calculation. On average, GIA takes 70.2 seconds to find the Pareto front of SAS and 228.8 minutes to find the Pareto front of Web Portal. For E-shop, GIA finds an average of only one Pareto-optimal solution in six days.

5.3 Performance Comparison

In a first set of experiments, we compare the performance of our parallel algorithms. We aim at determining which algorithm has the best performance, is the most scalable, and is the most promising to solve large MOCO problems such as E-shop. To obtain results in reasonable time, we use at most 16 processors simultaneously and apply each algorithm only to SAS and Web Portal.

Results. Figure 7 presents the speedups of each parallel algorithm finding the Pareto fronts of SAS and Web Portal using up to 16 processors. FS-GIA achieves super-linear speedups when using two processors for SAS and when using 2 to 16 processors for Web Portal. Furthermore, the speedup of FS-GIA increases steadily and rapidly when the number of available processors increases.

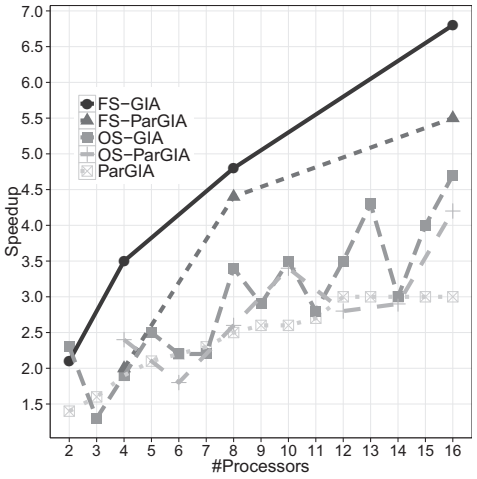
FS-ParGIA acquires super-linear speedups when using 8 to 16 processors for Web Portal. Using the same number of processors, the speedup of FS-ParGIA is less than that of FS-GIA. Also, the speedup of FS-ParGIA has a stable increasing trend with the increasing number of processors, but the speedup increasing rate of FS-ParGIA is lower than that of FS-GIA.

For ParGIA, OS-GIA, and OS-ParGIA, we did not observe super-linear or linear speedups in either case study. The speedup of ParGIA increases steadily and slowly as the number of processors increases, and then reaches a plateau (e.g., when using 12 or more processors for SAS, or using 5 or more processors for Web Portal).

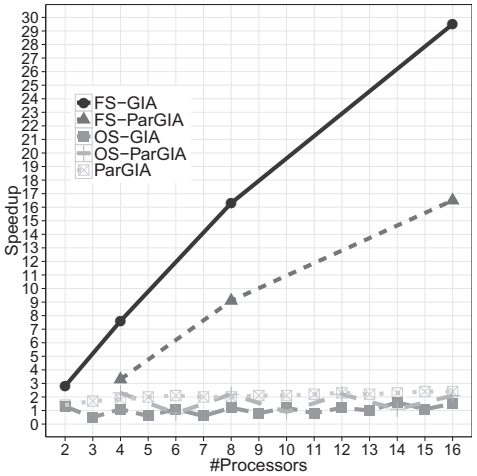
The speedups of OS-GIA and OS-ParGIA fluctuate considerably. In some cases, the speedups of OS-GIA and OS-ParGIA are less than one, which means that the execution time of OS-GIA and OS-ParGIA is even longer than the execution time of the sequential algorithm GIA.

Discussion. FS-GIA relies on a sound and efficient #SAT solver to control the overhead of dividing a given MOCO problem into subproblems and to guarantee that all subproblems are as equal-sized as possible. In the search space of each subproblem, the number of solutions that FS-GIA ex-

[8] `http://www.sharcnet.ca`.



(a) SAS



(b) Web Portal

Figure 7: Speedups of five parallel algorithms finding the Pareto fronts of (a) SAS and (b) Web Portal, using up to 16 processors

plores has been reduced substantially, which results in lower workload of conquering each subproblem. We are aware that equal-sized subproblems may not be equally *hard* [19], that is, solving each equal-sized subproblems may still need different workload. However, our empirical results demonstrate that FS-GIA effectively balances the overhead of the divide-and-conquer scheme and gains even super-linear speedups. In both case studies, FS-GIA achieves the best performance. More importantly, FS-GIA presents a desirable scalability when the number of available processors increases from 2 to 16, and the size of case studies augments from SAS to Web Portal.

FS-ParGIA uses the same divide-and-conquer approach as FS-GIA. For each divided subproblem, FS-ParGIA performs a ParGIA using two processors, while FS-GIA runs a GIA using one processor. When using the same number of processors, FS-GIA works more efficiently than FS-ParGIA. However, when addressing the same number of subproblems, FS-ParGIA usually works more efficiently than FS-GIA. For example, working with the same 4 subproblems of Web Portal, FS-ParGIA using 8 processors gains a higher speedup than FS-GIA using 4 processors. Therefore, there is a trade-off of using more processors for divide-and-conquer or for collaborative communication in each subproblem. Our empirical results show that it is worthwhile to spare more processors to perform more equal-sized subproblems instead of more GIA instances in each subproblem, especially when the number of available processors is limited.

ParGIA implements collaborative communication using message passing. However, due to the latency of message passing, a processor may already start searching an overlapping area before receiving the information about that area from other processors. The problem of overlapping searches may get worse when using more processors. According to our empirical results, ParGIA suffers from a performance bottleneck when running a large number of GIA instances simultaneously, and its speedup does not scale well with the increasing number of processors.

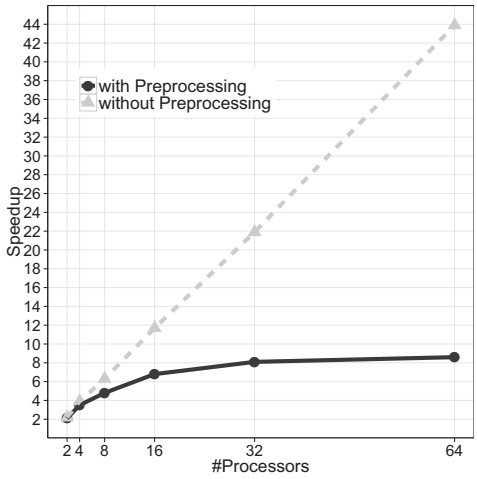
OS-GIA and OS-ParGIA geometrically divide the search space of a given MOCO problem into cones by projective-plane selection and cone separation. However, it is hard to foresee how solutions are distributed in the global search space and in the projective plane. Hence, the algorithms cannot guarantee that the workload of conquering each cone (i.e., finding Pareto-optimal solutions in each cone) is effectively balanced among processors, even though the cost of dividing the original MOCO problem (i.e., obtaining cones) is trivial. Unbalanced loads among processors gives rise to performance fluctuations when using OS-GIA and OS-ParGIA, as demonstrated by our experiments. In the worst case, finding the local Pareto front in a cone may take more time than finding the global Pareto front in the entire search space. This can happen when the number of locally Pareto-optimal solutions in a cone is higher than the number of globally Pareto-optimal solutions in the entire search space.

5.4 Scalability of FS-GIA

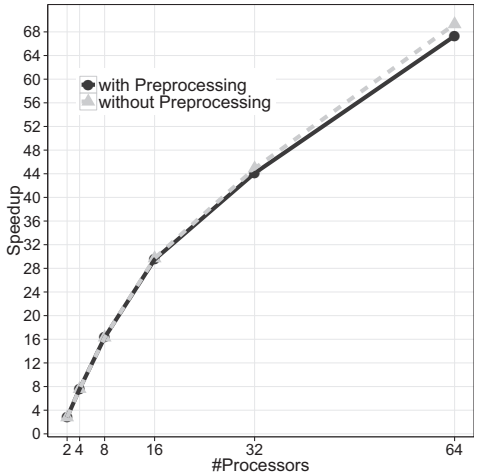
In a second series of experiments, we further explore the scalability of FS-GIA, as it performed best in the first set of experiments. We applied FS-GIA to all three case studies, using up to 64 processors simultaneously. We measured the preprocessing time of FS-GIA (i.e., the time of dividing a given MOCO problem into roughly equal-sized subproblems). Then, we analyzed the impact of the preprocessing time on the scalability of FS-GIA.

Results. Figure 8a shows the speedups of FS-GIA finding the Pareto front of SAS using up to 64 processors. For the normal execution time including the preprocessing (solid line), we observe that FS-GIA has a stable increasing trend of speedups when using 2 to 64 processors, but the increasing rate significantly slows down when using more than 32 processors. In contrast, if we ignore the preprocessing time (dashed line), then the speedup of FS-GIA maintains a stable and rapid increasing trend with the increasing number of processors.

For Web Portal, as shown in Figure 8b, the preprocessing time has little impact on the speedup of FS-GIA. Regardless of whether the preprocessing time is included in the execution time of FS-GIA, FS-GIA reaches super-linear speedups that scale well to any number of processors from 2 to 64.



(a) SAS



(b) Web Portal

Figure 8: Speedups of FS-GIA (with and without preprocessing) using up to 64 processors to find the Pareto front for SAS and Web Portal

For E-Shop, FS-GIA finds more Pareto-optimal solutions when using more processors. This is in contrast to the single Pareto-optimal solution found by the sequential algorithm GIA over the course of six days. If we determine the speedup in terms of the number of Pareto-optimal solutions found in six days, as shown in Figure 9, FS-GIA still provides super-linear speedups in all cases (ranging from 2 to 64 processors). Furthermore, the preprocessing time has almost no impact on the speedups of FS-GIA.

Discussion. The preprocessing time of FS-GIA reflects the overhead of dividing a given MOCO problem into subproblems. As explained in Section 3.3, the preprocessing time of FS-GIA increases logarithmically with the number of generated subproblems (i.e., equal to P where P is a power of two). Our empirical results show that we can effectively control the dividing overhead of FS-GIA. As listed in Table 3, the preprocessing of FS-GIA takes 6.6, 6.0, and 34.8 seconds to generate 64 equal-sized subproblems for SAS, Web Portal, and E-Shop.

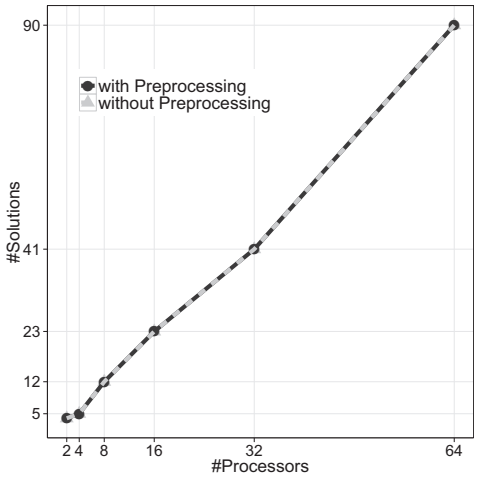


Figure 9: Number of Pareto-optimal solutions found by FS-GIA (with and without preprocessing) using up to 64 processors in six days for E-Shop

For small MOCO problems, such as SAS, each divided subproblem is small enough that it can be solved in seconds. In this case, the impact of the preprocessing time on the speedup of FS-GIA is non-negligible. As listed in Table 3, FS-GIA takes 6.6 seconds to divide the MOCO problem of SAS into 64 subproblems, whereas all subproblems can be solved simultaneously in 1.6 seconds. Thus, 80.5% of the execution time of FS-GIA is spent on the preprocessing stage. However, for large MOCO problems, such as Web Portal or E-Shop, the cost of conquering each subproblem is far more than the cost of dividing these MOCO problems into subproblems. In this case, the preprocessing time of FS-GIA has little impact on the speedups of FS-GIA, and the speedup of FS-GIA scales well with the increasing number of processors.

5.5 Threats to Validity

To increase internal validity, we use either standard or straightforward techniques to implement our algorithms. We chose an exact, general-purpose algorithm GIA as a baseline to solve MOCO problems sequentially. We rely on the standard solver sharpSAT to choose appropriate features for FS-GIA and FS-ParGIA generating equal-sized subproblems. In ParGIA, we implement the collaborative communication between processors using simple message passing. In OS-GIA and OS-ParGIA, we use an intuitive heuristic to choose two objectives with the first and second largest value ranges to form a large projective plane for cone separation. However, we cannot guarantee that the Pareto-optimal solutions are distributed evenly among the cones, and thus the parallelism in OS-GIA and OS-ParGIA suffers from unbalanced workloads among the processors.

To avoid the misleading effects caused by random fluctuation in measurements, we executed each algorithm 10 times on each case study, for each hardware configuration from 2 to 64 processors, and we used the means of the measured execution times in our analyses of speedups and scalability.

To help ensure external validity, we evaluated our proposed algorithms on three relatively large case studies from the literature. The three case studies cover a reasonable

Table 3: Impact of the preprocessing time of FS-GIA ($T_{FS-GIA_{prep}}$) on the entire execution time of FS-GIA (T_{FS-GIA}) and speedups; P − the number of processors; s − seconds; m − minutes

P	SAS				Web Portal				E-Shop
	T_{FS-GIA}(s)	$T_{FS-GIA_{prep}}$(s)	$\frac{T_{FS-GIA_{prep}}}{T_{FS-GIA}}$	Speedups	T_{FS-GIA}(m)	$T_{FS-GIA_{prep}}$(m)	$\frac{T_{FS-GIA_{prep}}}{T_{FS-GIA}}$	Speedups	$T_{FS-GIA_{prep}}$(s)
2	32.7	1.1	3.4 %	2.1	81.9	0.02	0.02 %	2.8	5.8
4	20.1	2.2	10.9 %	3.5	30.1	0.03	0.10 %	7.6	11.6
8	14.5	3.3	22.8 %	4.8	14.1	0.05	0.35 %	16.3	17.4
16	10.4	4.4	42.3 %	6.8	7.8	0.07	0.90 %	29.5	23.2
32	8.7	5.5	63.2 %	8.1	5.2	0.08	1.54 %	44.1	29.0
64	8.2	6.6	80.5 %	8.6	3.4	0.10	2.94 %	67.3	34.8

spectrum of MOCO problems with different characteristics, such as different sizes of problems, different types of quality attributes, and different optimization directions. One system has been used in a real-world scenario, and the other two systems are from a popular repository used by many researchers. However, we are aware that the results of our experiments may not transfer to other systems. We expect that especially large systems, such as Linux Kernel with thousands of features [36], would benefit from a combination of our parallel algorithms and other methods, such as approximation MOCO approaches.

6. RELATED WORK

In the field of Search Based Software Engineering (SBSE), multi-objective optimization has been identified as a major challenge for many software-engineering problems [20]. Metaheuristics, such as multi-objective evolutionary algorithms, have been used to provide approximate solutions for MOCO problems [37]. However, it is non-trivial for these metaheuristics to guarantee the accuracy of approximate solutions. On the one hand, metaheuristics depend mainly on a number of heuristically-chosen metrics, such as Hypervolume Indicator [46] and Maximum Spread [45], to evaluate the accuracy of approximate solutions. However, every metric provides some specific, but incomplete, quantifications of accuracy and can only be used effectively under certain conditions [43]. On the other hand, metaheuristics usually suffer from parameter sensitivity [18]. They often demand a considerable time to tune parameters for finding reasonably approximate solutions [32]. These problems motivated us to explore the feasibility of exact MOCO methods and improve their performance as far as possible.

As the size and complexity of a software system increases, not only exact methods but also metaheuristics became too time-consuming to address large MOCO problems [1, 36]. Many parallel models for metaheuristics have been proposed to solve MOCO problems efficiently, and they have been evaluated on a wide range of academic and real-world MOCO problems in different domains [7, 39]. Furthermore, parallelization of exact combinatorial-optimization methods, such as Branch and Bound [28] and Dynamic Programming [4], has been studied and implemented in multi-core environments [9]; however, it has been rarely addressed in the context of multi-objective optimization [39]. The only work we are aware is from Dhaenens et al. [13], who parallelized the exact solving of MOCO problems by geometrically splitting the search space into cubes and evaluated their algorithm on one case study; but their parallelization is not able to scale well up to only 10 processors. Thus, we propose a different geometric decomposition to partition the search space into cones. Moreover, we further scale the exact solving of MOCO problems by collaborative communication and equal-sized partition using #SAT. Our parallel algorithms target all MOCO problems, as long as each solution can be abstracted as a combination of features.

7. CONCLUSION

MOCO has been used to solve many problems in software engineering (e.g., architecture design [1], test data generation [23, 44], and project planning [20]) and other domains (e.g., hybrid vehicle powertrain design [34], electric vehicle battery design [25], and civil infrastructure repair planning [22]). We proposed five novel parallel algorithms for exact and efficient solving of MOCO problems. Our algorithms search for Pareto-optimal solutions using off-the-shelf solvers and parallelize the search via collaborative communication and divide-and-conquer. We conducted a series of experiments on three case studies of software-system design, covering a reasonable spectrum of MOCO problems with different characteristics. Our empirical results demonstrate the feasibility and performance of our parallel algorithms.

The key finding from our experiments is that FS-GIA outperforms all other proposed algorithms. FS-GIA partitions a given MOCO problem into subproblems of relatively equal size, which effectively balances the workload among the parallel processes. The result is that FS-GIA can achieve superlinear speedups. Moreover, the speedup of FS-GIA scales well up to 64 processors, and possibly beyond.

Our work opens a new direction in scaling exact MOCO methods. We hope that our work encourages other researchers to reconsider the feasibility of exact MOCO methods and to try different ways to scale them. Appropriate parallelization, especially given the increasing availability of multi-core systems, is definitely a promising approach.

In future, we plan to further improve our parallel algorithms and evaluate them on larger case studies. We expect that a combination of our exact, parallel algorithms with existing approximate approaches would further improve the accuracy and performance of solving MOCO problems.

8. ACKNOWLEDGMENTS

This work has been partially supported by NECSIS, Ontario Research Fund - Research Excellence Project on Model-Based Development, and the German Research Foundation (AP 206/4, AP 206/5, AP 206/6, and AP206/7).

9. REFERENCES

[1] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software Architecture Optimization Methods: A Systematic Literature Review. *IEEE Transactions on Software Engineering*, 39(5):658–683, 2013.

[2] G. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin-Cummings, 1989.

[3] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. SPLC*, pages 7–20. Springer, 2005.

[4] R. Bellman. *Dynamic Programming*. Dover, 2003.

[5] D. Benavides, S. Segura, and A. Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010.

[6] C. Blum and A. Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[7] J. Branke, H. Schmeck, K. Deb, and M. Reddy. Parallelizing Multi-Objective Evolutionary Algorithms: Cone Separation. In *Proc. CEC*, pages 1952–1957. IEEE, 2004.

[8] M. Carlsson, G. Ottosson, and B. Carlson. An Open-Ended Finite Domain Constraint Solver. In *Proc. PLILP*, pages 191–206. Springer, 1997.

[9] I. Chakroun, N. Melab, M.-S. Mezmaz, and D. Tuyttens. Combining Multi-Core and GPU Computing for Solving Combinatorial Optimization Problems. *J. Parallel and Distributed Computing*, 73(1):1563–1577, 2013.

[10] C. Coello, C. Dhaenens, and L. Jourdan. Multi-Objective Combinatorial Optimization: Problematic and Context. In *Advances in Multi-Objective Nature Inspired Computing*. Springer, 2010.

[11] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[12] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Comm. ACM*, 54(9):69–77, 2011.

[13] C. Dhaenens, J. Lemesre, and E. Talbi. K-PPM: A New Exact Method to Solve Multi-Objective Combinatorial Optimization Problems. *European Journal of Operational Research*, 200(1):45–53, 2010.

[14] N. Esfahani, S. Malek, and K. Razavi. GuideArch: Guiding the Exploration of Architectural Solution Space under Uncertainty. In *Proc. ICSE*, pages 43–52. IEEE, 2013.

[15] M. Gavanelli. An Algorithm for Multi-Criteria Optimization in CSPs. In *Proc. ECAI*, pages 136–140. IOS, 2003.

[16] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wąsowski. Variability-Aware Performance Prediction: A Statistical Learning Approach. In *Proc. ASE*, pages 301–311. IEEE, 2013.

[17] J. Guo, J. White, G. Wang, J. Li, and Y. Wang. A Genetic Algorithm for Optimized Feature Selection with Resource Constraints in Software Product Lines. *J. Systems and Software*, 84(12):2208–2221, 2011.

[18] D. Hadka and P. Reed. Diagnostic Assessment of Search Controls and Failure Modes in Many-Objective Evolutionary Optimization. *Evolutionary Computation*, 20(3):423–452, 2012.

[19] Y. Hamadi and C. M. Wintersteiger. Seven Challenges in Parallel SAT Solving. *AI Magazine*, 34(2):99–106, 2013.

[20] M. Harman. The Current State and Future of Search Based Software Engineering. In *Proc. FOSE*, pages 342–357. IEEE, 2007.

[21] M. Harman, S. Mansouri, and Y. Zhang. Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications. Technical report, King's College London TR-09-03, 2009.

[22] T. Hegazy and A. Elhakeem. Multiple Optimization and Segmentation Technique (MOST) for Large-Scale Bilevel Life Cycle Optimization. *Canadian Journal of Civil Engineers*, 38:263–271, 2011.

[23] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Multi-Objective Test Generation for Software Product Lines. In *SPLC*, pages 62–71. ACM, 2013.

[24] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT, 2006.

[25] A. Jarrett. *Multi-Objective Design Optimization of Electric Vehicle Battery Cooling Plates Considering Thermal and Pressure Objective Functions*. Master thesis, Queen University, 2011.

[26] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, CMU SEI, SEI-90-TR-21, 1990.

[27] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Chew. Optimistic Parallelism Requires Abstractions. In *Proc. PLDI*, pages 211–222. ACM, 2007.

[28] A. Land and A. Doig. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):497–520, 1960.

[29] M. Lukasiewycz, M. Glaß, C. Haubelt, and J. Teich. Solving Multi-Objective Pseudo-Boolean Problems. In *Proc. SAT*, pages 56–69. Springer, 2007.

[30] V. Marek and M. Truszczynski. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic programming paradigm: A 25-Year Perspective*. Springer, 1999.

[31] M. Mendonca, M. Branco, and D. Cowan. S.P.L.O.T. - Software Product Lines Online Tools. In *Proc. OOPSLA Companion*, pages 761–762. ACM, 2009.

[32] R. Olaechea. *Comparison of Exact and Approximate Multi-Objective Optimization for Software Product Lines*. Master thesis, University of Waterloo, 2013.

[33] D. Rayside, H.-C. Estler, and D. Jackson. A Guided Improvement Algorithm for Exact, General Purpose, Many-Objective Combinatorial Optimization. Technical report, MIT-CSAIL-TR-2009-033, 2009.

[34] J. Ribau, J. Sousa, and C. Silva. Multi-Objective Optimization of Fuel Cell Hybrid Vehicle Powertrain Design - Cost and Energy. Technical report, SAE 2013-24-0082, 2013.

[35] P. Saadatpanah, M. Famelis, J. Gorzny, N. Robinson, M. Chechik, and R. Salay. Comparing the Effectiveness of Reasoning Formalisms for Partial Models. In *Proc. MoDeVVa*, pages 41–46. ACM, 2012.

[36] A. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Scalable Product Line Configuration: A Straw to Break the Camel's Back. In *Proc. ASE*, pages 465–474. IEEE, 2013.

[37] A. Sayyad, T. Menzies, and H. Ammar. On the Value of User Preferences in Search-Based Software Engineering: A Case Study in Software Product Lines. In *Proc. ICSE*, pages 492–501. IEEE, 2013.

[38] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting Performance via Automated Feature-Interaction Detection. In *Proc. ICSE*, pages 167–177. IEEE, 2012.

[39] E.-G. Talbi, S. Mostaghim, T. Okabe, H. Ishibuchi, G. Rudolph, and C. Coello. Parallel Approaches for Multiobjective Optimization. In *Multiobjective Optimization*. Springer, 2008.

[40] M. Thurley. sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In *Proc. SAT*, pages 424–429. Springer, 2006.

[41] E. Tsang. *Foundations of Constraint Satisfaction*. Academic, 1993.

[42] D. Westermann, J. Happe, R. Krebs, and R. Farahbod. Automated Inference of Goal-Oriented Performance Prediction Functions. In *Proc. ASE*, pages 190–199. ACM, 2012.

[43] G. Yen and Z. He. Performance Metric Ensemble for Multiobjective Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 18(1):131–144, 2014.

[44] S. Yoo and M. Harman. Pareto Efficient Multi-Objective Test Case Selection. In *ISSTA*, pages 140–150. ACM, 2007.

[45] E. Zitzler, K. Deb, and L. Thiele. Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary Computation*, 8(2):173–195, 2000.

[46] E. Zitzler and L. Thiele. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.