An Approach for Non-domination Level Update Problem in Steady-State Evolutionary Algorithms With Parallelism

Sumit Mishra[1,2] and Carlos A. Coello Coello[3]

[1]Department of Computer Science & Engineering,
Indian Institute of Information Technology Guwahati, Assam – 781015, INDIA
[2]Departamento de Computación, CINVESTAV-IPN, Mexico City, MEXICO
[3]Departamento de Sistemas, UAM-Azcapotzalco, Mexico City, MEXICO
Email: sumit@iiitg.ac.in, ccoello@cs.cinvestav.mx

***Abstract*—One of the bottlenecks in steady-state multi-objective evolutionary algorithms (MOEAs) is non-dominated sorting because it is performed every time whenever a new offspring is generated. The recent literature shows that there is no requirement to perform the complete non-dominated sorting procedure because the entire structure of non-domination level (NDL) does not change. Some approaches have been recently proposed based on this idea. In this paper, we update our previous work where an offspring is inserted into the set of fronts, to further reduce the number of dominance comparisons. Additionally, we also explore parallelism in the updated approach in two different manners considering the PRAM CREW model. Finally, the time and space complexities of two parallel versions is theoretically analyzed.**

***Index Terms*—Evolutionary multi-objective optimization (EMO) algorithms, Non-domination level (NDL), Non-dominated sorting, Parallelism.**

I. Introduction

In the past decade, evolutionary algorithms have attracted a lot of attention because of their applicability in solving different problems with multiple objectives [1], [2]. Evolutionary algorithms which are used to solve multi-objective optimization problems (MOOPs) are termed as Multi-Objective Evolutionary Algorithms (MOEAs). Based on their selection scheme, MOEAs can be classified in two groups: (1) generational and (2) steady-state [3], [4]. In case of generational MOEAs, a set of offspring solutions is generated which may compete just among themselves or also with their parent solutions. However, in steady-state MOEAs, when a new offspring solution is generated, the parent population is updated to accommodate the new offspring solution. So, in case of steady-state MOEAs, the non-domination level (NDL) structure of the solutions changes when a new offspring is generated. The working flow of a steady-state MOEA is summarized in Algorithm 1. Here, when a new offspring solution is generated, the full non-dominated sorting [1] process needs to be performed. Let the initial population be $\mathbb{P} = \{sol_1, sol_2, \ldots, sol_N\}$ which contains N solutions and each solution has M objectives. Non-dominated sorting classifies these N solutions into K fronts which are arranged in decreasing order of their dominance $\mathcal{F} = \{F_1, F_2, \ldots, F_K\}$ such that

- All the solutions of a particular front are non-dominated with each other.
- Every solution of a front is dominated by one of the solutions in its previous front.

The cardinality of the front F_k is n_k, so $N = n_1 + n_2 + \ldots + n_K$. A solution is said to be non-dominated with a front F, if it is non-dominated with respect to all the solutions of F.

Algorithm 1 Working Flow of Steady-State EMO Algorithm

Input: Parameters of the algorithm
Output: Population $\mathbb{P}$
1: Population initialization $\mathbb{P} \leftarrow \{sol_1, sol_2, \ldots, sol_N\}$
2: **while** terminating condition is not satisfied **do**
3: $sol_{\text{off}} \leftarrow$ Generate an offspring solution
4: $\mathbb{P} \leftarrow \mathbb{P} \cup \{sol_{\text{off}}\}$
5: Apply non-dominated sorting on population $\mathbb{P}$ and obtain different fronts
6: Find the worst solution sol_{worst} from the sorted fronts
7: $\mathbb{P} \leftarrow \mathbb{P} \setminus \{sol_{\text{worst}}\}$
8: **return** $\mathbb{P}$

One simple solution to change the NDL structure is to apply non-dominated sorting considering the new offspring solution along with the existing solutions. A number of approaches have been proposed in the past for this purpose. The worst case time complexity of the naive approach [5] is $\mathcal{O}(MN^3)$ and the best case time complexity is $\mathcal{O}(MN^2)$. Deb *et al.* [1] developed an approach to improve the performance of the naive approach, with worst case time complexity $\mathcal{O}(MN^2)$. A divide-and-conquer based approach was proposed by Jensen *et al.* [6] with a time complexity $\mathcal{O}(N \log^{M-1} N)$. A divide-and-conquer based approach was also proposed by Fang *et al.* [7] with a worst case time complexity $\mathcal{O}(MN^2)$ and a best case time complexity $\mathcal{O}(MN \log N)$. This approach does not handle duplicate solutions correctly as one duplicate solution is considered as dominated by another. An approach based on

arena's principle was developed by Tang *et al.* [8] with best case time complexity $\mathcal{O}(MN\sqrt{N})$ [9].

Deductive and Climbing sort were developed by McClymont *et al.* [10]. These approaches infer the dominance relationship between two solutions in order to reduce the number of dominance comparisons. The worst case time complexity of Deductive and Climbing sort is $\mathcal{O}(MN^2)$. However, Deductive sort performs better than Climbing sort. The best case time complexity of Deductive sort is $\mathcal{O}(MN\sqrt{N})$. Wang *et al.* [11] proposed corner sort specially for many-objective cases. This approach has a worst case time complexity $\mathcal{O}(MN^2)$. The limitation of Jensen's approach was removed by Fortin *et al.* [12] with a worst case time complexity $\mathcal{O}(MN^2)$. Buzdalov *et al.* [13] proved the time complexity to be $\mathcal{O}(N\log^{M-1}N)$.

An efficient non-dominated sorting (ENS) approach was developed by Zhang *et al.* [9]. This approach works in two phases. Solutions are sorted based on a particular objective in the first phase. In the second phase, the solutions are taken from this sorted order to assign them to different fronts. The sorting in the first phase helps in reducing many unnecessary dominance comparisons. Two variants called ENS-SS and ENS-BS, are proposed. The worst case time complexity of both variants is $\mathcal{O}(MN^2)$. The best case time complexity of ENS-SS is $\mathcal{O}(MN\sqrt{N})$ and for ENS-BS is $\mathcal{O}(MN\log N)$. A divide-and-conquer based approach was proposed by Mishra *et al.* [14], [15] also with a best case time complexity $\mathcal{O}(MN\log N)$. Recently, Bao *et al.* [16] proposed a Hierarchical Non-dominated Sorting (HNDS) approach. The worst case time complexity of HNDS is $\mathcal{O}(MN^2)$ and its best case time complexity is $\mathcal{O}(MN\sqrt{N})$.

Best Order Sort (BOS) was proposed by Roy *et al.* [17]. BOS requires a lower number of dominance comparisons to sort the solutions. Mishra *et al.* [18] has generalized BOS by retaining the comparison set concept while handling duplicate solutions efficiently. The authors of BOS have recently proposed an improved version of BOS namely Bounded Best Order Sort (BBOS) [19]. A tree-based approach called T-ENS was proposed by Zhang *et al.* [20]. The best case time complexity of T-ENS is $\mathcal{O}(MN\log N/\log M)$. The worst case time complexity of T-ENS is $\mathcal{O}(MN^2)$. Based on ENS-BS [9], a tree-based approach called ENS-NDT was developed by Gustavsson *et al.* [21]. This approach was developed to significantly reduce the number of dominance comparisons performed even though its worst case time complexity is $\mathcal{O}(MN^2)$. This approach handles duplicate solutions efficiently.

Some approaches for non-dominated sorting focus on their efficient parallel implementation. These approaches have explored the parallelism in fast non-domitaed sorting [1]. A very fast non-dominated sorting procedure was proposed by Smutnicki *et al.* [22]. The time complexity of this parallel approach was proved to be $\mathcal{O}(M+N)$. Gupta *et al.* [23] proposed a parallel version which adopts GPUs. Three parallel versions were proposed by Ortega *et al.* [24] which are based on the use of multicores, GPUs and their combination. The parallelsim in BOS [17] has been analyzed in [25] considering multicore and GPUs. The parallelism in ENS [9] has been theoretically analyzed by Mishra *et al.* [26].

Algorithm 2 Steady-State EMO Algorithm to change the NDL structure

Input: Parameters of the algorithm
Output: Population $\mathbb{P}$

```
Population initialization ℙ ← {sol_1, sol_2, ..., sol_N}
Perform non-dominated sorting on ℙ
while stopping condition is not satisfied do
    sol_off ← Generate an offspring solution
    ℙ ← ℙ ∪ {sol_off}
    Insert sol_off in existing non-dominated fronts using our approach
    Find the worst solution sol_worst
    ℙ ← ℙ \ {sol_worst}
    Remove sol_worst from the existing non-dominated fronts
return ℙ
```

The incorporation of an offspring solution in the population does not change the entire NDL structure of the solutions [4], [27]–[29]. So, it becomes unnecessary to perform the complete non-dominated sorting again. Several approaches [4], [27], [29] have been proposed where the NDL structure is changed without performing the complete non-dominated sorting. In these approaches, the new offspring solution is inserted into the existing set of non-dominated fronts. The working flow of a steady-state MOEA which uses these approaches is summarized in Algorithm 2. In this algorithm, the set of solutions is sorted using non-dominated sorting at once. When a new offspring solution is generated, it is inserted into the existing set of fronts without performing the complete non-dominated sorting. Drozdik *et al.* [30] developed an M-front based approach with worst case time complexity $\mathcal{O}(MN^2)$. Buzdalov *et al.* [27] proposed an approach which is only suitable for 2 objectives. Mishra *et al.* [28] developed a dominance matrix based approach to restrict multiple dominance comparisons between the same pair of solutions in different generations of steady-state MOEAs. Li *et al.* [4] developed a generic approach which is suitable for any number of objectives. The worst case time complexity of this approach is $\mathcal{O}(MN^2)$, however, the maximum number of dominance comparisons is $\frac{1}{4}N^2$. The time complexity in the best case is $\mathcal{O}(M)$ which is a significant improvement over the best case time complexity of non-dominated sorting approaches. A tree based approach was proposed by Mishra *et al.* [29]. Another approach developed by Yakupov *et al.* [31] has a time complexity $\mathcal{O}(N\log^{M-2}N)$.

In this paper, we focus on updating the NDL structure of the solutions after the insertion of an offspring solution. Special attention is given to reduce the number of dominance comparisons. For this purpose, we have updated the approach proposed in [4], [29]. Although the worst case time complexity of the proposed approach is same as reported in [4], [29], the number of dominance comparisons may be reduced. Our proposed approach considers the sorted order of the existing solutions based on the first objective and this sorted order

of the solutions is used to reduce the number of dominance comparisons. The parallel version of the proposed approach is also explored considering the PRAM CREW model as this model is the earliest as well as one of the best-known model of parallel computation. In this regard, we have focused on the parallelism in two different manners and the time complexity of these two versions is also analyzed.

The rest of this paper is organized as follows. Our proposed approach is described in Section II. Its parallel version as well as its complexity analysis are discussed in Section III. Another parallel version with an improved time complexity is illustrated in Section IV. Finally, Section V concludes the paper and provides some possible paths for future research.

II. Our Proposed Approach

This section describes the procedure to insert an offspring solution sol_{off} in the sorted set of fronts $\mathcal{F} = \{F_1, F_2, \ldots, F_K\}$ which are organized in decreasing order of their dominance. In this approach, we are considering that the initial set of fronts is obtained by ENS [9], DCNS [14], [15], ENS-NDT [21] or T-ENS [20]. The reason is that these approaches initially sort the solutions based on an individual objective (generally the first objective) and we require this sorted order. We can also use any other sorting algorithms. However, in that case, we need to sort all the solutions based on an individual objective. This sorting is performed only once.

The procedure to insert an offspring solution into the set of fronts which are arranged in decreasing order of their dominance, is summarized in Algorithm 3. The input to this procedure is the solution to be inserted sol_{off}, the set of fronts $\mathcal{F} = \{F_1, F_2, \ldots, F_K\}$ and the sorted list of solutions based on the first objective denoted as *sL*. Here, we first find the correct position of sol_{off} in the sorted set of fronts, then the position of solutions in the existing set of fronts are updated whenever required. To find the proper position of sol_{off}, this solution is compared with the solutions of every front in $\mathcal{F}$ starting from the first front to the last until its proper position has been identified. Two solutions *sol* and *sol′* are compared using DomRelation(*sol*, *sol′*) procedure. This procedure can return any of the following three values:

- 0: When *sol* and *sol′* both are non-dominated.
- 1: When *sol* dominates *sol′*.
- -1: When *sol* is dominated by *sol′*.

When sol_{off} is compared with a solution of front $F_k (1 \leq k \leq K)$, then there can be three possibilities:

i. If sol_{off} is non-dominated with respect to a particular solution of F_k (line 6), then sol_{off} is compared with other solutions of F_k.

ii. If sol_{off} is dominated by a particular solution of F_k (line 8), then sol_{off} is compared with the solutions of the next front F_{k+1} as sol_{off} cannot be inserted into F_k. If sol_{off} is dominated by minimum one solution from all the existing K fronts (line 26), then sol_{off} is inserted into new front F_{K+1}.

Algorithm 3 Insert($\mathcal{F}, sol_{\text{off}}, sL$)

Input: $\mathcal{F} = \{F_1, F_2, \ldots, F_K\}$, sol_{off}: An offspring solution which is to be inserted into $\mathcal{F}$, *sL*: Sorted solutions based on the first objective
Output: Updated set of fronts when sol_{off} is inserted

```
1: for k ← 1 to K do
2:   ndCount ← 0
3:   S_dom ← ∅                          // Set of dominated solutions by sol_off
4:   for j ← 1 to |F_k| do
5:     domRel ← DomRelation(sol_off, F_k(j))      // Find the dominance relation between sol_off and F_k(j)
6:     if domRel = 0 then                // sol_off and F_k(j) are non-dominated
7:       ndCount ← ndCount + 1
8:     else if domRel = −1 then          // sol_off is dominated by F_k(j)
9:       Break                           // Check the next front for inclusion of sol_off
10:    else                              // sol_off dominates F_k(j)
11:      S_dom ← S_dom ∪ {F_k(j)}         // Add solution to S_dom
12:      F_k ← F_k \ {F_k(j)}             // Remove solution from F_k
13:      j ← j − 1
14:  if ndCount = |F_k| then              // sol_off is non-dominated with every solution of F_k
15:    F_k ← F_k ∪ {sol_off}              // Add sol_off to F_k
16:    return F                          // Process of sol_off insertion completes
17:  else if |F_k| = 0 then               // All the solutions of F_k are dominated by sol_off
18:    F_k ← F_k ∪ {sol_off}              // Add sol_off to F_k
19:    Increase the NDL of F_{k+1}, F_{k+2}, ..., F_K by 1
20:    F_{k+1} ← S_dom
21:    return F                          // Process of sol_off insertion completes
22:  else
23:    F_k ← F_k ∪ {sol_off}              // Add sol_off to F_k
24:    Update(F, k + 1, S_dom, sL)
25:    return F                          // Process of sol_off insertion completes
26: F_{K+1} ← sol_off                    // Create a new front
27: return F                             // Process of sol_off insertion completes
```

iii. If sol_{off} dominates a particular solution (line 10), then we continue comparing sol_{off} with the rest of the solutions in F_k to obtain the set of solutions in F_k which are dominated by sol_{off}. The solutions which are dominated by sol_{off} are stored in S_{dom} and removed from F_k.

If sol_{off} is non-dominated with respect to F_k (line 14), then sol_{off} is inserted into F_k and the process terminates. If sol_{off} dominates all the solutions in F_k (line 17), then NDL of the fronts $F_{k+1}, F_{k+2}, \ldots, F_K$ is increased by one and the solutions in S_{dom} are assigned to front F_{k+1}. Otherwise, *i.e.*, sol_{off} dominates some of the solutions in front $F_k (1 \leq k \leq K)$ (line 22), then solutions in the lower dominance front than F_k can be re-arranged. For this purpose, we use the recursive Update() procedure as discussed in Algorithm 4.

The set of solutions are re-arranged in lower dominance fronts than the front where sol_{off} is inserted, using the Update() procedure. In this procedure, we check for the solutions in F_{index} which are non-dominated with S. The non-dominated solutions from F_{index} are added to S and removed from F_{index}. By removing the solutions from F_{index}, a solution does not occupy more than one front. Let the initial cardinality of S be l. In the Update() procedure, the solutions of F_{index}

Algorithm 4 UPDATE($\mathcal{F}$, index, S, *sL*)

Input: $\mathcal{F} = \{F_1, F_2, \ldots, F_K\}$, index: NDL of the front whose solutions need to be checked for insertion in S, S: Set of solutions, *sL*: Sorted solutions based on the first objective in $\mathcal{F}$
Output: Updated set of non-dominated fronts $\mathcal{F}$

1: **if** index $= |\mathcal{F}| + 1$ **then**
2: $F_{|\mathcal{F}|+1} \leftarrow S$ *// S is the lowest dominance front*
3: **else**
4: $l \leftarrow |S|$ *// Store the initial cardinality of S*
5: **for** $i \leftarrow 1$ to $|F_{\text{index}}|$ **do** *// Consider each solution of F_{index} sequentially*
6: *check* $\leftarrow 0$
7: **for** $j \leftarrow 1$ to l **do** *// Check for comparison with initial l solutions of S*
8: **if** $sL[F_{\text{index}}(i)] > sL[S(j)]$ **then** *// Check whether $sL[F_{index}(i)]$ comes later in sL than $S(j)$*
9: *domRel* $\leftarrow$ DOMRELATION($S(j), F_{\text{index}}(i)$)
10: **if** *domRel* $= 1$ **then** *// $S(j)$ dominates $F_{index}(i)$*
11: *check* $\leftarrow 1$
12: **BREAK**
13: **if** *check* $= 0$ **then** *// $F_{index}(i)$ is non-dominated with first l solutions of S*
14: $S \leftarrow S \cup \{F_{\text{index}}(i)\}$ *// Add $F_{index}(i)$ to S*
15: $F_{\text{index}} \leftarrow F_{\text{index}} \setminus \{F_{\text{index}}(i)\}$ *// Remove $F_{index}(i)$ from F_{index}*
16: $i - -$
17: **if** $l = |S|$ **then** *// No solution from F_{index} is non-dominated with every solution of S*
18: Increase the NDL of $F_{\text{index}}, F_{\text{index}+1}, \ldots, F_K$ by 1
19: $F_{\text{index}} \leftarrow S$
20: **else if** $F_{\text{index}} = \emptyset$ **then** *// Each of the solutions from F_{index} has been added to S*
21: $F_{\text{index}} \leftarrow S$
22: **else**
23: $T \leftarrow F_{\text{index}}$
24: $F_{\text{index}} \leftarrow S$
25: UPDATE($\mathcal{F}$, index $+ 1, T, sL$)

are compared with the initial l solutions of S because when a solution of F_{index} is found to be non-dominated with respect to S, then it is added to S thus increasing its cardinality. In general, a solution of F_{index} is compared with all the solutions in S before being inserted in S as in [4], [29]. In this approach, we are focusing on reducing the number of dominance comparisons. So, for this purpose, a solution of F_{index} is not compared with all the solutions in S. Instead, this solution is only compared with those solutions in S which occur before that solution in the sorted list *sL*. The reason for this is that the solution in F_{index} cannot dominate the solutions in S as in the initial set of fronts the solutions in F_{index} belong to a lower dominance front than that of S. Also, the solution which comes later in *sL* cannot dominate the solution which comes earlier in this list. So, the solution in F_{index} which comes earlier than the solution of S in *sL* will always be non-dominated with respect to the solution of S. Thus, we compare only those solutions in F_{index} which come later than the solution of S in *sL* because these are the only solutions which can be dominated by the solutions of S. After comparing the solution of F_{index} with the solutions of S, one of the following three conditions may occur:

- No solution from F_{index} is found to be non-dominated with respect to S, *i.e.*, $l = |S|$ (line 17). In this case, the NDL of fronts $F_{\text{index}}, F_{\text{index}+1}, \ldots, F_K$ is increased by 1. Also, the non-domination level 'index' is assigned to S, *i.e.*, S will constitute front F_{index}.
- All the solutions of F_{index} are added to S, *i.e.*, $F_{\text{index}} = \emptyset$ (line 20). In this case, S will constitute front F_{index}.
- Otherwise (line 22), we move all the solutions of F_{index} to T. The non-domination level of F_{index} is assigned to S, *i.e.*, S will constitute front F_{index}. Now, the update procedure is repeated with UPDATE($\mathcal{F}$, index $+ 1, T, sL$).

Implication of having the sorted order of the solutions: We have used the sorted solutions based on the first objective to reduce the number of dominance comparisons. However, these sorted solutions do not always help to reduce the number of dominance comparisons. This depends on the orientation of the solutions in the fronts. If the values of the first objective of the solutions in lower dominance fronts overlap with the values of the first objective of the solutions in higher dominance fronts, then the number of dominance comparisons may be reduced. If the values of the first objective of the solutions in lower dominance fronts is larger than the values of the first objective of the solutions in higher dominance fronts, then the number of dominance comparisons will remain the same as the one obtained in [4], [29].

A. Space Complexity

In this approach, we are storing the sorted order of the solutions based on the first objective. So, the space required by our approach is $\mathcal{O}(N)$ considering that there are N solutions in the initial population.

B. Time Complexity

In this approach, we first find the position of the offspring solution in the sorted list of solutions based on the first objective. The time complexity of obtaining the position of the offspring solution using a binary search strategy is $\mathcal{O}(\log N)$ considering that there are N solutions in the sorted list.

We first discuss the time complexity of the UPDATE() procedure. Let the offspring solution sol_{off} be inserted into the k^{th} front. In the worst case, each call to the UPDATE() procedure moves one solution from F_{index} to its preceding front. In this case, the maximum number of dominance comparisons is given by Eq. (1). For the number of dominance comparison to be maximum, an offspring solution dominates $n_k - 1$ solutions in front F_k before being inserted in that front.

$$\#dComp_{\text{Update}_{\text{general\_max}}} = (n_k - 1)\, n_{k+1} + (n_{k+1} - 1)\, n_{k+2} + \ldots + (n_{K-1} - 1)\, n_K \quad (1)$$

However, overall, the maximum number of dominance comparisons is given by Eq. (2). In this case, the offspring solution

is inserted into the first front and dominates $n_1 - 1$ solutions in that front.

$$\#dComp_{\text{Update}_{\text{overall\_max}}} = (n_1 - 1)\, n_2 + (n_2 - 1)\, n_3 + \ldots + (n_{K-1} - 1)\, n_K \quad (2)$$

Hence, the worst case time complexity of the UPDATE() procedure is $\mathcal{O}(MN^2)$. The minimum number of dominance comparisons occurs when the UPDATE() procedure is called with the index value $K + 1$, *i.e.*, the offspring solution sol_{off} is inserted into the last front F_K. In this case, the number of dominance comparisons is given by Eq. (3) and the time complexity in the best case of the UPDATE() procedure is $\mathcal{O}(1)$.

$$\#dComp_{\text{Update}_{\text{min}}} = 0 \quad (3)$$

In this approach, initially the position of the offspring solution sol_{off} is identified in the sorted set of fronts. Let the offspring solution sol_{off} be inserted into the k^{th} front F_k. After the position is identified, the solutions in the set of fronts with a lower dominance than F_k are checked for their movement in the next higher dominance front. Thus, the maximum number of dominance comparisons is obtained by Eq. (4) when the offspring solution sol_{off} is inserted into front F_k after dominating $n_k - 1$ solutions in that front.

$$\begin{aligned}\#dComp_{\text{general}} &= (n_1 + n_2 + \ldots + n_k) + \#dComp_{\text{Update}_{\text{general\_max}}} \\ &= (n_1 + n_2 + \ldots + n_k) + [(n_k - 1)\, n_{k+1} + \\ &\quad (n_{k+1} - 1)\, n_{k+2} + \ldots + (n_{K-1} - 1)\, n_K]\end{aligned} \quad (4)$$

However, overall, the maximum number of dominance comparisons is obtained by Eq. (5). Thus, the worst case time complexity of our approach is $\mathcal{O}(MN^2)$.

$$\begin{aligned}\#dComp_{\text{overall}} &= (n_1) + \#dComp_{\text{Update}_{\text{overall\_max}}} \\ &= (n_1) + [(n_1 - 1)\, n_2 + (n_2 - 1)\, n_3 + \ldots + \\ &\quad (n_{K-1} - 1)\, n_K]\end{aligned} \quad (5)$$

The number of dominance comparisons becomes maximum when there are two fronts [4], [29]. When the number of solutions N is even, the cardinality of the first front, *i.e.*, $n_1 = {}^{N}/_{2} + 1$ and the cardinality of second front, *i.e.*, $n_2 = {}^{N}/_{2} - 1$. When the number of solutions N is odd, the cardinality of the first front, *i.e.*, $n_1 = \lceil {}^{N}/_{2} \rceil$ and the cardinality of the second front, *i.e.*, $n_2 = \lfloor {}^{N}/_{2} \rfloor$. The maximum number of dominance comparisons for an even number of solutions is ${}^{N^2}/_{4} + 1$ and for an odd number of solutions is ${}^{(N^2+3)}/_{4}$ [4], [29].

III. PARALLEL ALGORITHM-1

The main motivation for parallelizing evolutionary algorithms is that their implementation has a low data dependency. This section describes a parallel algorithm to update the NDL structure of the solutions after insertion of an offspring solution. In the UPDATE() procedure of the serial algorithm, all the solutions of a particular front are checked to see whether they can be inserted into their immediately higher dominance front or not. In particular, the solutions of F_{index} are checked for their insertion in S. A particular solution of F_{index} can be simultaneously compared with l solutions of S. The solution in F_{index} which is non-dominated with respect to S is added to S and removed from F_{index}. The parallel version of the UPDATE() procedure is summarized in Algorithm 5.

Algorithm 5 UPDATE PARALLEL-1($\mathcal{F}$, index, S, sL)

Input: Same as Algorithm 4
Output: Updated set of non-dominated fronts $\mathcal{F}$

1: **if** index $= |\mathcal{F}| + 1$ **then**
2: $F_{|\mathcal{F}|+1} \leftarrow S$ // *S is the lowest dominance front*
3: **else**
4: $l \leftarrow |S|$ // *Store the initial cardinality of S*
5: **for** $i \leftarrow 1$ to $|F_{\text{index}}|$ **do**
6: *isDominated*$[1, 2, \ldots, l] \leftarrow$ FALSE // *Stores whether a particular solution of F_{index} is dominated by the solutions of S or not*
/* PARALLEL SECTION STARTS */
7: **for** $j \leftarrow 1$ to l **do**
8: **if** $sL[F_{\text{index}}(i)] > sL[S(j)]$ **then** // *Check whether $F_{index}(i)$ comes later in sL than $S(j)$*
9: *isDominated*$[j] \leftarrow F_{\text{index}}(i)$ is dominated by $S(j)$ or not
/* PARALLEL SECTION ENDS */
10: Process *isDominated*[ ] array in a parallel manner to see wether solution $F_{\text{index}}(i)$ is non-dominated with respect to S or not
11: **if** $F_{\text{index}}(i)$ is non-dominated with respect to S **then**
12: $S \leftarrow S \cup \{F_{\text{index}}(i)\}$ // *Add $F_{index}(i)$ to S*
13: $F_{\text{index}} \leftarrow F_{\text{index}} \setminus \{F_{\text{index}}(i)\}$ // *Remove $F_{index}(i)$ from F_{index}*
14: $i - -$
15: **if** $|S| = l$ **then** // *No solution from F_{index} is non-dominated with respect to S*
16: Increase the NDL of $F_{\text{index}}, F_{\text{index}+1}, \ldots, F_K$ by 1
17: $F_{\text{index}} \leftarrow S$
18: **else if** $F_{\text{index}} = \emptyset$ **then** // *Each of the solutions from F_{index} has been added to S*
19: $F_{\text{index}} \leftarrow S$
20: **else**
21: $T \leftarrow F_{\text{index}}$
22: $F_{\text{index}} \leftarrow S$
23: UPDATE PARALLEL-1($\mathcal{F}$, index $+ 1$, T, sL)

As the initial cardinality of S is l, an array of size l denoted as *isDominated*[ ] is created which stores whether a particular solution $sol \in F_{\text{index}}$ is dominated by a solution $s \in S$. 'TRUE' in this array means that *sol* is dominated and 'FALSE' indicates that it is not dominated. After comparing *sol* with the solutions of S in a parallel manner and storing their dominance relation in an array *isDominated*[ ], this array is processed in a parallel manner to know whether *sol* is non-dominated with respect to S or not.

To process *isDominated*[ ] in a parallel manner various '*OR*' operations are performed at $\log l$ levels. At the x^{th} level, ${}^{l}/_{2^x}$ '*OR*' operations are performed. All the '*OR*' operations at each level can be performed simultaneously. After performing the '*OR*' operation at the last level, a final value is obtained. If we get 'TRUE', then it means that a particular solution

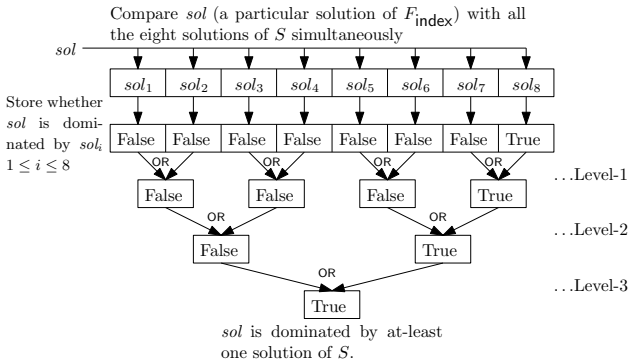

Fig. 1: Simultaneous comparison of a solution $sol \in F_{\text{index}}$ with the solutions in S and checking whether sol is non-dominated with respect to S.

sol is dominated by at least one solution of S, otherwise sol is not dominated by any of the solutions, *i.e.*, it is non-dominated and can be added to S. The time complexity of processing *isDominated*[ ] in a parallel manner is $\mathcal{O}(\log l)$ as the '*OR*' operation are performed at $\log l$ levels and all the '*OR*' operations at a level are performed simultaneously.

Example 1. *Let $S = \{sol_1, sol_2, \ldots, sol_8\}$. Consider there is a solution sol of F_{index}. Solution sol is compared with $sol_1, sol_2, \ldots, sol_8$ simultaneously. An array of size 8 is used which stores whether sol is dominated by $sol_i (1 \le i \le 8)$. This array of size eight is processed in a parallel manner at three levels. Two consecutive values in this array are processed using an 'OR' operation. At the end, we obtain 'TRUE' which means that sol is dominated by at least one of the solutions of S. This complete process is shown in Fig. 1.*

The *isDominated*[ ] array can be filled in $\mathcal{O}(M)$ time in a parallel manner. Let the offspring solution sol_{off} be inserted into the k^{th} front. Then, the worst case time complexity is given by Eq. (6). Here, the offspring solution dominates $n_k - 1$ solutions in the k^{th} front.

$$T_{\text{update}} = [M + \log(n_k - 1)\, n_{k+1}] + [M + \log(n_{k+1} - 1)\, n_{k+2}] + \ldots + [M + \log(n_{K-1} - 1)\, n_K] \quad (6)$$

However, overall, the worst case time complexity of the parallel version of the UPDATE() procedure occurs when the offspring solution is inserted into F_1 and is given by Eq. (7).

$$T_{\text{update}} = [M + \log(n_1 - 1)\, n_2] + [M + \log(n_2 - 1)\, n_3] + \ldots + [M + \log(n_{K-1} - 1)\, n_K] \quad (7)$$

The worst case time complexity to update the NDL structure using the parallel version of the UPDATE() procedure is given by Eq. (8) as the number of fronts is two in the worst case [4], [29]. In this Equation, the $\mathcal{O}(\log N)$ factor represents the time to find the position of the offspring solution in the sorted list of solutions based on the first objective.

$$\begin{aligned} T_{\text{approach}} &= \log N + Mn_1 + [M + \log(n_1 - 1)\, n_2] \\ &= \log N + Mn_1 + M + n_2 \log(n_1 - 1) \\ &= \mathcal{O}(MN + N \log N) \end{aligned} \quad (8)$$

When the dominance nature of each solution with respect to other solutions are obtained in a parallel manner in $\mathcal{O}(M)$ time beforehand and stored in a matrix as in [22], then the time complexity in the worst case is obtained by Eq. (9).

$$\begin{aligned} T_{\text{approach}} &= \log N + M + n_1 + [1 + \log(n_1 - 1)\, n_2] \\ &= \log N + M + n_1 + 1 + n_2 \log(n_1 - 1) \\ &= \mathcal{O}(M + N \log N) \end{aligned} \quad (9)$$

Algorithm 6 UPDATE PARALLEL-2($\mathcal{F}$, index, S, sL)

Input: Same as Algorithm 4
Output: Updated set of non-dominated fronts $\mathcal{F}$

1: **if** index $= |\mathcal{F}| + 1$ **then**
2: $F_{|\mathcal{F}|+1} \leftarrow S$ // *S is the lowest dominance front*
3: **else**
4: $l \leftarrow |S|$ // *Store the initial cardinality of S*
5: *isNonDominated*$[1, 2, \ldots, |F_{\text{index}}|] \leftarrow$ FALSE // *Stores whether a particular solution of F_{index} is non-dominated with respect to all the solutions of S or not*
/* PARALLEL SECTION STARTS */
6: **for** $i \leftarrow 1$ to $|F_{\text{index}}|$ **do**
7: *isDominated*$[1, 2, \ldots, l] \leftarrow$ FALSE // *Stores whether a particular solution of F_{index} is dominated by the solutions of S or not*
/* PARALLEL SECTION STARTS */
8: **for** $j \leftarrow 1$ to l **do**
9: **if** $sL[F_{\text{index}}(i)] > sL[S(j)]$ **then** // *Check whether $F_{index}(i)$ comes later in sL than $S(j)$*
10: *isDominated*$[j] \leftarrow F_{\text{index}}(i)$ is dominated by $S(j)$ or not
/* PARALLEL SECTION ENDS */
11: Process *isDominated*[ ] array in parallel manner to know wether $F_{\text{index}}(i)$ is non-dominated with respect to S. If it is non-dominated then make *isNonDominated*$[i]$ to be TRUE, *i.e.*, *isNonDominated*$[i] \leftarrow$ TRUE
/* PARALLEL SECTION ENDS */
12: **for** $i \leftarrow 1$ to $|F_{\text{index}}|$ **do**
13: **if** *isNonDominated*$[i] =$ TRUE **then**
14: $S \leftarrow S \cup \{F_{\text{index}}(i)\}$ // *Add $F_{index}(i)$ to S*
15: $F_{\text{index}} \leftarrow F_{\text{index}} \setminus \{F_{\text{index}}(i)\}$ // *Remove $F_{index}(i)$ from F_{index}*
16: $i - -$
17: **if** $|S| = l$ **then** // *No solution from F_{index} is non-dominated with every solution of S*
18: Increase the NDL of $F_{\text{index}}, F_{\text{index}+1}, \ldots, F_K$ by 1
19: $F_{\text{index}} \leftarrow S$
20: **else if** $F_{\text{index}} = \emptyset$ **then** // *Each of the solutions from F_{index} has been added to S*
21: $F_{\text{index}} \leftarrow S$
22: **else**
23: $T \leftarrow F_{\text{index}}$
24: $F_{\text{index}} \leftarrow S$
25: UPDATE PARALLEL-2($\mathcal{F}$, index $+ 1, T, sL$)

IV. Parallel Algorithm-2

In the first parallel version of the UPDATE() procedure, a solution of F_{index} is compared with all the solutions of S simultaneously. However, all the solutions of F_{index} are processed in a sequential manner. To further speed up the UPDATE() procedure, all the solutions in F_{index} can be processed simultaneously. If any of these solutions are non-dominated with respect to each of the solutions of S, then these solutions are added to S in a sequential manner to avoid a write collision. The parallel version of the improved UPDATE() procedure is summarized in Algorithm 6. In this parallel version, an array of size l corresponding to each solution of F_{index} is considered which stores whether that particular solution of F_{index} is dominated by the solutions of S or not. By processing this array in a parallel manner, it can be decided whether the particular solution of F_{index} can be added to S or not. Here, an array *isNonDominated*[ ] of length $|F_{\text{index}}|$ is considered which stores whether a particular solution of F_{index} is non-dominated with respect to S or not. This information is obtained by processing the *isDominated*[ ] array for that particular solution in a parallel manner.

Example 2. *Let's assume that there are four solutions* $\{sol_1, sol_2, sol_3, sol_4\}$ *in* S *and* F_{index} *also has four solutions* $\{sol'_1, sol'_2, sol'_3, sol'_4\}$*. All the four solutions of* F_{index} *are simultaneously compared with respect to the four solutions of* S *in a parallel manner. Each solution of* F_{index} *is checked to see whether it is dominated by at least one of the solutions of* S *or not. After finding the dominance relation of all the solutions of* F_{index} *with respect to* S*, if this solution is non-dominated then, it is added to* S*. This complete process is shown in Fig. 2.*

We discuss the time complexity of this second parallel version of the UPDATE() procedure. Let the offspring solution sol_{off} be inserted into the k^{th} front. Then, the worst case time complexity is given by Eq. (10). Here, the offspring solution dominates $n_k - 1$ solutions in the k^{th} front.

$$T_{\text{update}} = [M + \log(n_k - 1) + n_{k+1}] + [M + \log(n_{k+1} - 1) + n_{k+2}] + \ldots + [M + \log(n_{K-1} - 1) + n_K] \quad (10)$$

However, overall, the worst case time complexity of this second parallel version of the UPDATE() procedure occurs when the offspring solution is inserted into the first front and is given by Eq. (11).

$$T_{\text{update}} = [M + \log(n_1 - 1) + n_2] + [M + \log(n_2 - 1) + n_3] + \ldots + [M + \log(n_{K-1} - 1) + n_K] \quad (11)$$

The worst case time complexity to update the NDL structure using the parallel version of the UPDATE() procedure is given by Eq. (12) as the number of fronts is two in the worst case [4], [29].

$$\begin{aligned} T_{\text{approach}} &= \log N + Mn_1 + [M + \log(n_1 - 1) + n_2] \\ &= \mathcal{O}(MN) \end{aligned} \quad (12)$$

When the dominance nature of each solution with respect to other solutions are obtained in a parallel manner in $\mathcal{O}(M)$ time beforehand and stored in a matrix as in [22], then the time complexity in the worst case is obtained by Eq. (13).

$$\begin{aligned} T_{\text{approach}} &= \log N + M + n_1 + [1 + \log(n_1 - 1) + n_2] \\ &= \mathcal{O}(M + N) \end{aligned} \quad (13)$$

V. Conclusions & Future Work

In the current paper, we have focused on updating the NDL structure of the solutions when a new solution is inserted into the existing set of fronts. We have suggested a way to reduce the number of dominance comparisons. However, the maximum number of dominance comparisons remains the same as in the approaches proposed in [4], [29]. Our proposed approach considers the sorted order of the existing solutions based on the first objective and this sorted order of the solutions is used to reduce the number of dominance comparisons. To further improve the performance of updating the NDL structure, we have focused on its parallelization considering the PRAM CREW model. In this regard, we have explored the parallelism in two different manners. The time complexity in the worst case of the parallel version is $\mathcal{O}(M + N)$ which is the same as the time complexity of the parallel version of non-dominated sorting as reported in [22].

For future work, we are interested in improving the performance of the algorithm in terms of the dominance comparisons and, if possible, its worst case time complexity as well.

Acknowledgements

The second author is on sabbatical leave from CINVESTAV-IPN. He gratefully acknowledges support from CONACyT grant no. 2016-01-1920 (*Fronteras de la Ciencia 2016*).

References

[1] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA–II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, April 2002.

[2] K. Deb and H. Jain, "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, August 2014.

[3] J. J. Durillo, A. J. Nebro, F. Luna, and E. Alba, "On the Effect of the Steady-State Selection Scheme in Multi-Objective Genetic Algorithms," in *Evolutionary Multi-Criterion Optimization. 5th International Conference, EMO 2009*. April 2009, pp. 183–197.

[4] K. Li, K. Deb, Q. Zhang, and Q. Zhang, "Efficient Nondomination Level Update Method for Steady-State Evolutionary Multiobjective Optimization," *IEEE Transactions on Cybernetics*, vol. 47, no. 9, pp. 2838–2849, September 2017.

[5] N. Srinivas and K. Deb, "Multiobjective Optimization Using Nondominated Sorting in Genetic Algorithms," *Evolutionary Computation*, vol. 2, no. 3, pp. 221–248, Fall 1994.

[6] M. T. Jensen, "Reducing the Run-Time Complexity of Multiobjective EAs: The NSGA-II and Other Algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 5, pp. 503–515, October 2003.

[7] H. Fang, Q. Wang, Y.-C. Tu, and M. F. Horstemeyer, "An Efficient Non-dominated Sorting Method for Evolutionary Algorithms," *Evolutionary Computation*, vol. 16, no. 3, pp. 355–384, Fall 2008.

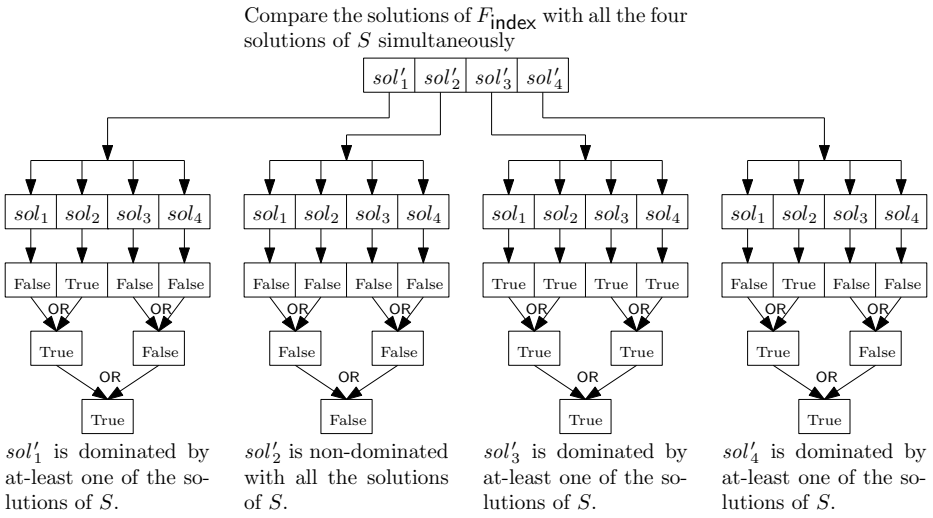

Fig. 2: Parallel Comparison of the solution in F_{index} with respect to the solutions in S and checking whether that solution is non-dominated with respect to all the solutions of S.

[8] S. Tang, Z. Cai, and J. Zheng, "A Fast Method of Constructing the Non-dominated Set: Arena's Principle," in *2008 Fourth International Conference on Natural Computation*. Jinan, China: IEEE Computer Society Press, 18-20 October 2008, pp. 391–395.

[9] X. Zhang, Y. Tian, R. Cheng, and J. Yaochu, "An Efficient Approach to Nondominated Sorting for Evolutionary Multiobjective Optimization," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 2, pp. 201–213, April 2015.

[10] K. McClymont and E. Keedwell, "Deductive Sort and Climbing Sort: New Methods for Non-dominated Sorting," *Evolutionary Computation*, vol. 20, no. 1, pp. 1–26, Spring 2012.

[11] H. Wang and X. Yao, "Corner Sort for Pareto-Based Many-Objective Optimization," *IEEE Transactions on Cybernetics*, vol. 44, no. 1, pp. 92–102, January 2014.

[12] F.-A. Fortin, S. Greiner, and M. Parizeau, "Generalizing the Improved Run-Time Complexity Algorithm for Non-dominated Sorting," in *2013 Genetic and Evolutionary Computation Conference (GECCO'2013)*. New York, USA: ACM Press, July 2013, pp. 615–622.

[13] M. Buzdalov and A. Shalyto, "A Provably Asymptotically Fast Version of the Generalized Jensen Algorithm for Non-dominated Sorting," in *Parallel Problem Solving from Nature - PPSN XIII, 13th International Conference*. September 13-17 2014, pp. 528–537.

[14] S. Mishra, S. Saha, and S. Mondal, "Divide and Conquer Based Non-dominated Sorting for Parallel Environment," in *2016 IEEE Congress on Evolutionary Computation (CEC'2016)*. Vancouver, Canada: IEEE Press, 24-29 July 2016, pp. 4297–4304.

[15] S. Mishra, S. Saha, S. Mondal, and C. A. Coello Coello, "A Divide-and-Conquer Based Efficient Non-dominated Sorting Approach," *Swarm and Evolutionary Computation*, vol. 44, pp. 748–773, February 2019.

[16] C. Bao, L. Xu, E. D. Goodman, and L. Cao, "A Novel Non-dominated Sorting Algorithm for Evolutionary Multi-Objective Optimization," *Journal of Computational Science*, vol. 23, pp. 31–43, November 2017.

[17] P. C. Roy, M. M. Islam, and K. Deb, "Best Order Sort: A New Algorithm to Non-dominated Sorting for Evolutionary Multi-objective Optimization," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. Denver, Colorado, USA: ACM Press, July 20-24 2016, pp. 1113–1120.

[18] S. Mishra, S. Mondal, S. Saha, and C. A. Coello Coello, "GBOS: Generalized Best Order Sort Algorithm for Non-dominated Sorting," *Swarm and Evolutionary Computation*, vol. 43, pp. 244–264, 2018.

[19] P. C. Roy, K. Deb, and M. M. Islam, "An Efficient Nondominated Sorting Algorithm for Large Number of Fronts," *IEEE Transactions on Cybernetics*, no. 99, pp. 1–11, 2018.

[20] X. Zhang, Y. Tian, R. Cheng, and Y. Jin, "A Decision Variable Clustering-Based Evolutionary Algorithm for Large-Scale Many-Objective Optimization," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 1, pp. 97–112, February 2018.

[21] P. Gustavsson and A. Syberfeldt, "A New Algorithm Using the Non-dominated Tree to Improve Non-dominated Sorting," *Evolutionary Computation*, vol. 26, no. 1, pp. 89–116, September 2018.

[22] C. Smutnicki, J. Rudy, and D. Zelazny, "Very Fast Non-dominated Sorting," *Decision Making in Manufacturing and Services*, vol. 8, no. 1-2, pp. 13–23, 2014.

[23] S. Gupta and G. Tan, "A Scalable Parallel Implementation of Evolutionary Algorithms for Multi-Objective Optimization on GPUs," in *2015 IEEE Congress on Evolutionary Computation (CEC'2015)*. Sendai, Japan: IEEE Press, 25-28 May 2015, pp. 1567–1574.

[24] G. Ortega, E. Filatovas, E. M. Garzon, and L. G. Casado, "Non-dominated Sorting Procedure for Pareto Dominance Ranking on Multicore CPU and/or GPU," *Journal of Global Optimization*, vol. 69, no. 3, pp. 607–627, November 2017.

[25] J. Moreno, G. Ortega, E. Filatovas, J. Martínez, and E. Garzón, "Improving the Performance and Energy of Non-dominated Sorting for Evolutionary Multiobjective Optimization on GPU/CPU Platforms," *Journal of Global Optimization*, pp. 1–19, 2018.

[26] S. Mishra and C. A. Coello Coello, "P-ENS: Parallelism in Efficient Non-Dominated Sorting," in *2018 IEEE Congress on Evolutionary Computation (CEC'2018)*. Rio de Janeiro, Brazil: IEEE Press, July 8–13 2018, pp. 508–515.

[27] M. Buzdalov, I. Yakupov, and A. Stankevich, "Fast Implementation of the Steady-State NSGA-II Algorithm for Two Dimensions Based on Incremental Non-dominated Sorting," in *2015 Genetic and Evolutionary Computation Conference (GECCO 2015)*. Madrid, Spain: ACM Press, July 11-15 2015, pp. 647–654.

[28] S. Mishra, S. Mondal, and S. Saha, "Fast Implementation of Steady-State NSGA-II," in *2016 IEEE Congress on Evolutionary Computation (CEC'2016)*. Vancouver, Canada: IEEE Press, 24-29 July 2016, pp. 3777–3784.

[29] S. Mishra, S. Mondal, and S. Saha, "Improved Solution to the Non-Domination Level Update Problem," *Applied Soft Computing*, vol. 60, pp. 336–362, November 2017.

[30] M. Drozdik, Y. Akimoto, H. Aguirre, and K. Tanaka, "Computational Cost Reduction of Nondominated Sorting Using the M-Front," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 5, pp. 659–678, October 2015.

[31] I. Yakupov and M. Buzdalov, "Improved Incremental Non-dominated Sorting for Steady-State Evolutionary Multiobjective Optimization," in *2017 Genetic and Evolutionary Computation Conference (GECCO'2017)*. Berlin, Germany: ACM Press, July 15-19 2017, pp. 649–656.