# A Divide and Conquer Approach for Web Services Location Allocation Problem

Harshal Tupsamudre*
harshalcoep@gmail.com

Saket Saurabh†
sk.saurabh@tcs.com

Arun Ramamurthy†
ramamurthy.arun@tcs.com

Mangesh Gharote†
mangesh.g@tcs.com

Sachin Lodha†
sachin.lodha@tcs.com

## ABSTRACT

The appropriate choice of locations for the deployment of web services is of significant importance. The placement of a web service closer to user centers minimizes the response time, however deployment cost may increase. The placement becomes more challenging when multiple web services are involved. In this paper, we address the problem of placing multiple web services with the aim of simultaneously minimizing conflicting objectives of total deployment cost and network latency. We solve the location allocation problem for each web service independently and combine the resulting solutions using a novel merge algorithm. We demonstrate through extensive experiments and simulations that the proposed approach is not only computationally efficient but also produces good quality solutions. Further, the proposed merge algorithm is generic and could be easily adapted to tackle any bi-objective optimization problem that can be decomposed into non-overlapping sub-problems.

## KEYWORDS

Web services; Multi-Objective Optimization; Location Allocation Problem; Divide and Conquer

**ACM Reference Format:**
Harshal Tupsamudre, Saket Saurabh, Arun Ramamurthy, Mangesh Gharote, and Sachin Lodha. 2021. A Divide and Conquer Approach for Web Services Location Allocation Problem. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3449726.3463127

## 1 INTRODUCTION

Service-oriented computing using web services has emerged as a new computing paradigm for developing software applications. These web services provide a distributed computing infrastructure for both intra- and cross-enterprise application integration and collaboration. As the number of functionally similar web services is steadily increasing, Quality of Service (QoS) (such as response time [1]) has become vital to gain a competitive advantage over other service providers [2].

Choosing appropriate web service locations to serve geographically distributed user centers significantly impacts QoS and customer satisfaction. Deploying a web service closer to each user center improves the service response time; however, setting up a web service at multiple locations invariably increases the deployment cost. There could also be multiple web services to be deployed. This problem of placing multiple web services to simultaneously minimize two conflicting objectives, namely total deployment cost and network latency, is known as Web Services Location Allocation Problem (WSLAP) [3].

WSLAP is a variant of a facility location problem which is proved to be NP-hard [4] [5]. The search space of WSLAP is combinatorial. If there are, say, $s$ web services and $n$ candidate locations, and each service can be deployed at multiple locations, then the number of solutions is $2^{s \times n}$ [3]. To find multiple Pareto-optimal solutions using exact approaches such as Integer Linear Programming (ILP), branch and bound algorithm etc., is challenging.

Therefore, researchers have explored various Multi-Objective Evolutionary Algorithms (MOEAs) such as MOPSO [6], NSGAII [7] and NSGA-II with a local search [8] in WSLAP. As MOEAs work with a population of solutions, they can produce a set of trade-off solutions in a few generations. Further, MOEAs scale much better than ILPs due to the nature of heuristic search. Recently, Tan et al. [3] proposed a Binary Multi-Objective Particle Swarm Optimization with Crowding Distance (BMOPSOCD) algorithm to solve WSLAP and showed that it produces better solutions than NSGAII. Many a time, decomposing the problem into smaller sub-problems, solving each sub-problem independently and merging is an efficient way to solve the problem [9]. The main contribution of this work is a novel merge algorithm to solve the WSLAP using a Divide and Conquer (D&C) approach. Particularly, we solve the location allocation problem for each web service using a MOEA (NSGAII or BMOPSOCD) and efficiently merge the resulting non-dominated solutions of each web service to obtain solutions for WSLAP. The major benefits of D&C are:

- Problem size reduces significantly.
- Sub-problems being independent can be solved in parallel.
- Sub-problems that are similar (in terms of configuration and requirements) need to be solved only once.

Further, the approach is incremental as it enables web service providers to find deployment locations for a new web service and

*Work was done as a part of TCS Research and Innovation
†TCS Research and Innovation, Tata Consultancy Services, India


*GECCO '21 Companion, July 10–14, 2021, Lille, France*

ACM ISBN 978-1-4503-8351-6/21/07…$15.00
https://doi.org/10.1145/3449726.3463127

perform allocation of user centers without having to solve the entire problem from scratch.

The merge algorithm takes non-dominated solutions of multiple sub-problems (web services) and produces the solution to the original problem (WSLAP). We conducted multiple experiments with different WSLAP instance sizes to demonstrate that the D&C approach is not only efficient but also produces better solutions in terms of quality as well as diversity than the state-of-the-art [3].

The organization of this paper is as follows. First, we briefly describe the multi-objective WSLAP model as given by Tan et al. [3]. Subsequently, we explain our D&C algorithm in detail. Later, we compare the efficacy of our D&C approach with the state-of-the-art MOEAs [3] on the original WSLAP model. Finally, we conclude the paper and discuss the future work.

## 2 WSLAP DESCRIPTION

The location allocation problem involving multiple web services [3] is described as follows. A set of $s$ web services $W = \{W_1, W_2, \ldots, W_s\}$ needs to be deployed at one or more of the $n$ candidate locations $A = \{A_1, A_2, \ldots, A_n\}$. The locations could be the data centers of the enterprise itself and/or one or more cloud providers. The cost incurred to deploy a web service $W_i$ at a location $A_j$ is given by $C_{ij}$. The deployment cost varies from one data center to another due to the difference in electricity price, real estate price, human labour cost etc. All web services are assumed to be independent of each other. If that is not the case, we can merge dependent services into a single web service.

There are $m$ user centers $U = \{U_1, U_2 \ldots U_m\}$ that require access to each of the $s$ web services. A user center represents a geographic region. It also allows us to estimate the latency between the user center and candidate locations. The service invocation frequency (demand) of a web service $W_i$ from a user center $U_k$ over a unit time interval is represented by $F_{ki}$. The service frequencies are computed as the average number of invocations over a period of time, such as over a month. The network latency between a user center $U_k$ and a location $A_j$ is given by $L_{kj}$. Each service has to be deployed in at least one location. Further, a service can be deployed at multiple locations to improve its response time. Note that the requirement of a user center pertaining to a given web service is catered by exactly one location. The matrices required for modelling the input in WSLAP are given in Table 1.

**Table 1: Input and Output Matrices in the WSLAP**

| Matrix | Entry | Description |
|---|---|---|
| $W_{s\times1}$ | $W_i$ | $i^{th}$ web service |
| $A_{n\times1}$ | $A_j$ | $j^{th}$ location |
| $U_{m\times1}$ | $U_k$ | $k^{th}$ user center |
| $F_{m\times s}$ | $F_{ki}$ | Frequency invocation of service $W_i$ at $U_k$ |
| $C_{s\times n}$ | $C_{ij}$ | Cost of deploying service $W_i$ at $A_j$ |
| $L_{m\times n}$ | $L_{kj}$ | Latency between $U_k$ and $A_j$ |
| $X_{s\times n}$ | $X_{ij}$ | Decision variable indicating whether $W_i$ is deployed at location $A_j$ |

The mathematical model of WSLAP is as follows:

$$minimize \quad f_1 = \sum_{i=1}^{s}\sum_{j=1}^{n} C_{ij} \cdot X_{ij} \tag{1}$$

$$minimize \quad f_2 = \sum_{k=1}^{m}\sum_{i=1}^{s} R_{ki} \cdot F_{ki} \tag{2}$$

*subject to*:

$$\sum_{j=1}^{n} X_{ij} \geq 1 \quad \forall i \in \{1, \ldots, s\} \tag{3}$$

$$X_{ij} \in \{0, 1\} \quad \forall i \in \{1, \ldots, s\} \quad \forall j \in \{1, \ldots, n\} \tag{4}$$

The objective of WSLAP is to locate a set of web services $W$ among a set of candidate locations $A$ such that the total deployment cost $f_1$ and the total response time $f_2$ are minimized. A service location matrix $X$ of size $s \times n$ is used to represent the entire location plan. The value of the decision variable $X_{ij}$ is 1 if web service $W_i$ is deployed at location $A_j$; otherwise, it is 0. A user center is served from the nearest web service location among the deployed locations to minimize latency.

To minimize latency, each user center is served from the nearest web service location among the set of deployed locations. $R_{ki}$ represents the minimal response time incurred when web service $W_i$ is accessed by user center $U_k$ and is calculated as follows.

$$R_{ki} = min\{L_{kj} \mid X_{ij} = 1 \ \& \ j \in \{1, \ldots, n\}\} \tag{5}$$

As the dimensions of the service location matrix $X$ is $s \times n$ and each entry takes a binary value 0 or 1, the size of the search space for WSLAP is $2^{s\times n}$.

A solution of WSLAP is a tuple $(a, b)$, where $a$ represents the total deployment cost ($f_1$) and $b$ represents the total latency ($f_2$). A solution $\theta_i = (a_i, b_i)$ is said to dominate solution $\theta_j = (a_j, b_j)$, if $\theta_i$ is at least as good as $\theta_j$ in one objective and strictly better in the other objective,

- $a_i < a_j$ and $b_i \leq b_j$ or
- $a_i \leq a_j$ and $b_i < b_j$

A solution $\theta^*$ is referred to as Pareto-optimal if it is not dominated by any other solution. A multi-objective optimization problem typically has many Pareto-optimal (non-dominated) solutions.

**Locations**: In the original problem, a web service can be deployed in at most $n$ locations. If the candidate locations are cloud regions, then the value of $n$ is typically 50 (e.g., Microsoft Azure). If multiple cloud providers are considered, then the value of $n$ easily exceeds 100. Solving WSLAP could generate many solutions where each service is deployed in a large number of locations. In practice, a web service provider might consider deploying its web service $W_i$ in at most $d_i$ locations ($d_i << n$). Therefore, to accomodate this upper bound $d_i$, we change the WSLAP constraint given in equation (3) as follows:

$$1 \leq \sum_{j=1}^{n} X_{ij} \leq d_i, \quad d_i \leq n \ \ \forall i \in \{1, \ldots, s\} \tag{6}$$

If we set $d_i$ to be $n$, then the problem becomes equivalent to the original WSLAP.

## 3 OUR APPROACH

We show that WSLAP with the aforementioned characteristics can be tackled efficiently using a D&C approach. The steps involved in our D&C approach is as follows:

1. Decompose WSLAP into $s$ subproblems, one for each web service.

2. Solve each subproblem independently using MOEAs.
3. Merge solutions of all subproblems.

In the rest of the section, first, we prove that WSLAP can be decomposed into $s$ sub-problems, where $s$ is the number of unique web services, and each sub-problem can be solved independently. Later, we give a novel space and time efficient merge algorithm for combining non-dominated solutions of each sub-problem to obtain non-dominated solutions for the original WSLAP problem.

THEOREM 1. *The multi-objective web service location allocation problem (WSLAP) comprising of s web services can be decomposed into s sub-problems, one corresponding to each web service.*

We can re-write the first objective of WSLAP given in equations (1) as follows:

$$f_1 = \Big(\sum_{j=1}^{n} C_{1j} \cdot X_{1j}\Big) + \Big(\sum_{j=1}^{n} C_{2j} \cdot X_{2j}\Big) + \ldots + \Big(\sum_{j=1}^{n} C_{sj} \cdot X_{sj}\Big) \quad (7)$$

$$= Cost\ of\ W_1 + Cost\ of\ W_2 + \ldots + Cost\ of\ W_s \quad (8)$$

$$= f_{1_1} + f_{1_2} + \ldots + f_{1_s} \quad (9)$$

where, $f_{1_i}$ denotes the deployment cost of web service $W_i$.

Similarly, we can re-write the second objective as follows:

$$f_2 = \Big(\sum_{k=1}^{m} R_{k1} \cdot F_{k1}\Big) + \Big(\sum_{k=1}^{m} R_{k2} \cdot F_{k2}\Big) + \ldots + \Big(\sum_{k=1}^{m} R_{ks} \cdot F_{ks}\Big) \quad (10)$$

$$= Latency\ of\ W_1 + Latency\ of\ W_2 + \ldots + Latency\ of\ W_s \quad (11)$$

$$= f_{2_1} + f_{2_2} + \ldots + f_{2_s} \quad (12)$$

where, $f_{2_i}$ denotes the latency of web service $W_i$.

Similarly, the constraint given in equation (6) that upper-bounds the number of deployment locations for each web service can be decomposed as follows:

$$1 \le \sum_{j=1}^{n} X_{1j} \le d_1, \ \ldots\ , \ 1 \le \sum_{j=1}^{n} X_{sj} \le d_s \quad (13)$$

Grouping the corresponding terms in equations (9), (12) and (13), we can model and solve the location allocation problem for each web service independently.

COROLLARY 1. *If a solution $(f_{1_i}, f_{2_i})$ for web service $W_i$ is not Pareto-optimal then the corresponding solution $(f_1, f_2)$ to the original problem WSLAP is also not Pareto-optimal.*

By Theorem 1, the solution $(f_1, f_2)$ to WSLAP is computed using solution obtained for each web service. Therefore, we have:

$$f_1 = f_{1_1} + f_{1_2} + \ldots + f_{1_i} + \ldots + f_{1_s}$$
$$f_2 = f_{2_1} + f_{2_2} + \ldots + f_{2_i} + \ldots + f_{2_s}$$

Since the solution $(f_{1_i}, f_{2_i})$ corresponding to web service $W_i$ is not Pareto-optimal, there exists a better solution $(f'_{1_i}, f'_{2_i})$ which dominates it. Assume that $f'_{1_i} \le f_{1_i}$ and $f'_{2_i} < f_{2_i}$. Therefore, the new solution $(f'_1, f'_2)$ to the original WSLAP problem can be computed as follows:

$$f'_1 = f_{1_1} + f_{1_2} + \ldots + f'_{1_i} + \ldots + f_{1_s} \le f_1$$
$$f'_2 = f_{2_1} + f_{2_2} + \ldots + f'_{2_i} + \ldots + f_{2_s} < f_2$$

As $(f_1, f_2)$ is dominated by $(f'_1, f'_2)$, it cannot be Pareto-optimal. Note: Similar argument can be made for the case of $f'_{1_i} < f_{1_i}$ and $f'_{2_i} \le f_{2_i}$

COROLLARY 2. *If the relative invocation frequency and the deployment cost of a web service $W_i$ is the same as that of another web service $W_g$, then the solution for $W_g$ can be obtained using solutions for $W_i$ and vice versa.*

Consider the relative invocation frequency of service $W_i$ for $m$ user centers, that is:

$$F_{1i} \Big/ \sum_{k=1}^{m} F_{ki}, F_{2i} \Big/ \sum_{k=1}^{m} F_{ki}, \ldots, F_{mi} \Big/ \sum_{k=1}^{m} F_{ki}$$

Similarly, consider the relative invocation frequency of service $W_g$:

$$F_{1g} \Big/ \sum_{k=1}^{m} F_{kg}, F_{2g} \Big/ \sum_{k=1}^{m} F_{kg}, \ldots, F_{mg} \Big/ \sum_{k=1}^{m} F_{kg}$$

We are given that the relative invocation frequency and cost of both services are the same. Hence, we have:

$$F_{k'i} \Big/ \sum_{k=1}^{m} F_{ki} = F_{k'g} \Big/ \sum_{k=1}^{m} F_{kg} \quad \forall k' \in \{1, \ldots, m\}$$

$$C_{ij} = C_{gj} \quad \forall j \in \{1, \ldots, n\}$$

Let $\sum_{k=1}^{m} F_{ki} = z \cdot \sum_{k=1}^{m} F_{kg}$

Therefore, we have:
$F_{ki} = z \cdot F_{kg} \quad \forall k \in \{1, \ldots, m\}$
Let $(f^*_{1g}, f^*_{2g})$ be a Pareto-optimal solution for service $W_g$. We claim that $(f^*_{1g}, z \cdot f^*_{2g})$ is a Pareto-optimal solution for service $W_i$. Suppose that another solution $(f_{1i}, f_{2i})$ of $W_i$ dominates solution $(f^*_{1g}, z \cdot f^*_{2g})$. Therefore, there exists a solution $(f_{1i}, f_{2i}/z)$ for $W_g$ that dominates solution $(f^*_{1g}, f^*_{2g})$. Hence, the contradiction.

Once a web service $W_i$ is located, allocation of user centers is relatively easy. Each user center is allocated to the nearest web service location. Therefore, search space for location allocation of a web service is $\sum_{l=1}^{d_i} \binom{n}{l}$. Since there are $s$ web services, the search space is at most $\sum_{i=1}^{s} \sum_{l=1}^{d_i} \binom{n}{l}$. After obtaining non-dominated solutions to each web service, we need to combine them to find the non-dominated solutions for the original problem. Let the set of non-dominated solutions for each web service $W_i$ be denoted by $S_i$. Therefore, the total solution space of WSLAP is:

$$search\ space = O\Big(\sum_{i=1}^{s} \sum_{l=1}^{d_i} \binom{n}{l}\Big) + \prod_{i=1}^{s} |S_i| \quad (14)$$

To summarize, since all web services are independent, we can divide WSLAP into $s$ smaller sub-problems, one corresponding to each

web service, find non-dominated solutions to each sub-problem (web service) and finally merge the solutions of all sub-problems to obtain solutions to the original problem (WSLAP). The major benefits of using D&C approach for WSLAP are as follows:

**1. Efficiency**: Each sub-problem is relatively smaller and independent, hence can be solved in parallel. The existing approaches proposed in the literature [6][7][3] do not exploit these characteristics and attempt to solve the problem in its entirety.

**2. Incremental**: If a new web service is added to the existing problem, then the previously proposed techniques [6][7][3] need to resolve the entire problem from scratch. With the D&C approach, we only need to find location-allocation for the new web service and merge the resulting solutions with the existing solutions. Thus, the proposed approach is incremental.

**3. Reuse**: In some applications, it is likely that the invocation frequency and deployment cost of few web services are the same. With the D&C approach, we only need to find solutions for one such web service (Corollary 2).

The pseudocode of our D&C approach is depicted in Algorithm 1. First, we obtain a set of non-dominated solutions $S_i$ for each web service $W_i$ using evolutionary algorithms described in the next section (lines 4-6). The resulting sets of non-dominated solutions $(S_1, S_2, \ldots, S_s)$ are then combined using the *Merge* procedure (line 7). The merge algorithm is iterative and starts by combining non-dominated solutions of the first two web services which are then combined with non-dominated solutions of the third web service and so on (lines 13-16). The *Combine* procedure consists of three main steps. First, it adds non-dominated solutions of two sub-problems $E_1$ and $E_2$ in all possible ways (lines 23-29). The resulting set of solutions $E = \{(a+c, b+d) \mid (a,b) \in E_1 \text{ and } (c,d) \in E_2\}$ is of size $|E_1| \cdot |E_2|$. Second, it sorts set $E$ by the first objective in ascending order. Third, it uses Kung's et al. *Front* method [10] which is a recursive procedure to find non-dominated solutions from the input set $E$.

**Algorithm 1** D&C Algorithm

```
1:  procedure DivideAndConquer
2:  Input: Frequency matrix F_{m×s}, Cost matrix C_{s×n} and Latency matrix L_{m×n}
3:  Output: A set of non-dominated solutions S to WSLAP
4:      for i = 1 to s do
5:          S_i = Solve(F[1 : m][i], C[i], L)
6:      end for
7:      Merge(S_1, S_2, ..., S_s)
8:  end procedure
9:
10: procedure Merge
11: Input: A set of non-dominated solutions S_i for each of the s web services
12: Output: A set of non-dominated solutions S to WSLAP
13:     S = S_1
14:     for i = 2 to s do
15:         S = Combine(S, S_i)
16:     end for
17:     return S
18: end procedure
19:
20: procedure Combine
21: Input: Two sets of non-dominated solutions E_1 and E_2
22: Output: A set of non-dominated solutions in the combined set E = {(a + c, b + d)|(a, b) ∈
    E_1 and (c, d) ∈ E_2}
23:     E = ⊥
24:     for x = 1 to |E_1| do
25:         for y = 1 to |E_2| do
26:             s = (a_x + c_y, b_x + d_y)
27:             E[(x − 1) ∗ |E_2| + y] = s
28:         end for
29:     end for
30:     sort tuples in E by first objective in ascending order
31:     return Front(E)
32: end procedure
```

Theorem 2. *The Merge procedure takes as input a list of non-dominated solution sets $S_1, S_2 \ldots, S_s$, where $S_i$ is a solution set for $i^{th}$ sub-problem, and computes the set of non-dominated solutions $S$ to the original problem.*

We prove the correctness of *Merge* procedure using a loop invariant [11]. The loop invariant is at the start of $i^{th}$ iteration where set $S$ contains non-dominated solutions to the first $i-1$ sub-problems.

**Initialization.** Before the loop starts, set $S$ is initialized to the set $S_1$ of non-dominated solutions for the first sub-problem (line 13). Hence, at the start of the loop, when the value of $i$ is 2, $S = S_1$. Therefore, the loop invariant initially holds.

**Maintenance.** Suppose that the invariant holds at the beginning of iteration $i$, that is, $S$ contains non-dominated solutions to the first $i-1$ sub-problems. The *Combine* procedure is called to merge solutions of the first $i-1$ sub-problems with solution set $S_i$ of the $i^{th}$ sub-problem (line 15). The procedure begins by adding non-dominated solutions in two sets $S$ and $S_i$ in all possible ways (lines 23-29) and creates a new set $E = \{(a+c, b+d) \mid (a,b) \in S \text{ and } (c,d) \in S_i\}$. Then, it sorts set $E$ by first objective in ascending order (line 30) and calls Kung's et al. *Front* method [10] to find the set of non-dominated solutions $N$ in $E$ (line 31). The set $N$ is returned and assigned to $S$. The set $S$ now contains non-dominated solutions to the first $i$ sub-problems and, therefore, the invariant holds at the end of the iteration as well.

**Termination.** The loop terminates when $i = s+1$. Thus, from the invariant, $S$ contain non-dominated solutions to all $s$ sub-problems and hence to the original problem.

We note that the *Combine* procedure given in Algorithm 1 requires $O(|E1| \cdot |E2|)$ space to store all possible solutions (lines 24-29) and $O(|E1| \cdot |E2| \cdot log(|E1| \cdot |E2|))$ time for sorting all possible solutions (line 30). However, the number of non-dominated solutions could be much less than $|E1| \cdot |E2|$. In the next section, we give a space and time efficient method to combine non-dominated solutions of two sub-problems.

## 3.1 Novel Combine Algorithm

Consider two sub-problems $P_1$ and $P_2$. Suppose that $P_1$ has $r$ unique non-dominated solutions $E_1 = \{(a_1, b_1), \ldots, (a_r, b_r)\}$ and $P_2$ has $t$ unique non-dominated solutions $E_2 = \{(c_1, d_1), \ldots, (c_t, d_t)\}$, where $r \leq t$. We consider only unique solutions to avoid redundant computations. Further, assume that each solution set is sorted in ascending (increasing) order by first objective i.e, $a_1 < a_2 < \ldots < a_r$ and $c_1 < c_2 < \ldots < c_t$. Adding solutions in sets $E_1$ and $E_2$ in all possible ways yields a solution set $E$ consisting of $r \cdot t$ tuples as shown in Figure 1.

Theorem 3. *Consider a solution tuple $(a_i + c_j, b_i + d_j) \in E$. It cannot dominate or be dominated by tuples $(a_k + c_l, b_k + d_l)$, where $(k \leq i \text{ and } l \leq j)$ or $(k \geq i \text{ and } l \geq j)$.*

Since all solutions in $E_1$ are unique, non-dominated and arranged in increasing order of the first objective $f_1$, they must be in decreasing order of the second objective $f_2$. Similar is the case with solutions in $E_2$.

Since $a_k < a_i$ and $b_k > b_i$ for $k < i$, and $c_l < c_j$ and $d_l > d_j$ for

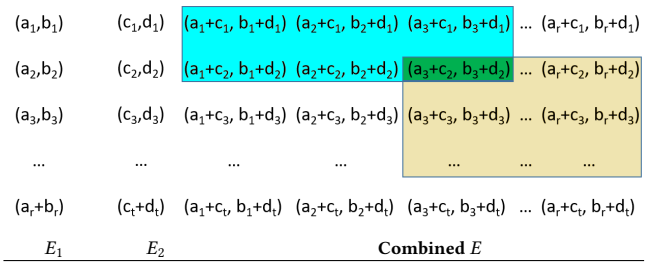


**Figure 1: Two sets $E_1$ and $E_2$ of non-dominated solutions containing $r$ and $t$ tuples respectively, and their combination $E$ consisting of $r \cdot t$ candidate solutions.**

$l < j$, we have

$$a_k + c_l < a_i + c_j, b_k + d_l > b_i + d_j \tag{15}$$

The above inequality holds even when $k = i$ (and $l < j$) or $l = j$ (and $k < i$). Hence, $(a_i + c_j, b_i + d_j)$ cannot dominate or be dominated by $(a_k + c_l, b_k + d_l)$, where $k \leq i$ and $l \leq j$.

Similarly, $a_k > a_i$ and $b_k < b_i$ for $k > i$, and $c_l > c_j$ and $d_l < d_j$ for $l > j$. Hence, we have

$$a_k + c_l > a_i + c_j, b_k + d_l < b_i + d_j \tag{16}$$

The above inequality holds even when $k = i$ (and $l > j$) or $l = j$ (and $k > i$). Hence, $(a_i + c_j, b_i + d_j)$ cannot dominate or be dominated by $(a_k + c_l, b_k + d_l)$, where $k \geq i$ and $l \geq j$. Thus, the proof.

Consider a solution $(a_3 + c_2, b_3 + d_2) \in E$ which is highlighted in green in Figure 1c. The theorem implies that all solution tuples that lie to the top-left (shaded in cyan) of $(a_3+c_2, b_3+d_2)$ are better in the first objective but worse in the second objective. Similarly, all solution tuples that lie to the bottom right (shaded in yellow) are better in the second objective but worse in the first objective. Therefore, $(a_3 + c_2, b_3 + d_2)$ need not be compared with these solutions.

Based on this observation, we propose an iterative procedure $CombineEff$ (Algorithm 2) which generates $|E| = r \cdot t$ solution tuples strategically to avoid unnecessary comparisons and returns a set of non-dominated solutions $N \subseteq E$. The strategy is to generate a solution tuple $(a_i+c_j, b_i+d_j)$ only if the left tuple $(a_{i-1}+c_j, b_{i-1}+d_j)$ and the top tuple $(a_{i+1} + c_{j-1}, b_{i+1} + d_{j-1})$ have already been generated and compared with the existing non-dominated solutions. The new procedure uses two data structures, min-heap $H$ and stack $N$. The generated solutions are stored temporarily in the min-heap. The solutions that are confirmed to be non-dominated are removed from the min-heap and stored in the stack. The value of first objective $f_1$ is used as key of min-heap. If two tuples have the same key ($f_1$ value), then they are compared using the second objective $f_2$. We imagine all possible solutions being arranged in a $t \times r$ table (as shown in Figure 1c) and identify each solution tuple using a row number and column number. An integer array $flag$ of size $r$ stores the row index of recently removed tuple (from min-heap) for each column.

Algorithm 2 proceeds as follows. Initially, the stack $N$ contains a dummy solution $(0, \infty)$ (lines 4-5). The first tuple $(a_1 + c_1, b_1 + d_1)$ is generated and added to the min-heap $H$ since its $f_1$ value is the lowest (lines 6-7). At the start of each iteration, the tuple $min = (f_1, f_2)$ is removed from the root node (line 10). Its $f_2$ value is compared with $f_2$ value of solution at the top of the stack $N$. If the value is smaller, then the solution $min$ is pushed in the stack (lines 11-13). Note that, solutions are removed from the heap in the increasing order of first objective $f_1$. If $f_1$ value of $min$ is equal to $f_1$ value of solution at the top of the stack, then $f_2$ value of $min$ must be less than the $f_2$ value of top solution.

Assume that the solution $min$ belongs to row $j$ and column $i$ in the solution table of size $t \times r$ (Figure 1c). The $i^{th}$ entry in $flag$ is updated with the row number $j$ since it is the most recent solution in column $i$ which is removed from the min-heap (line 16). Now, there is a possibility of generating two solutions, one that lies in the right column $i + 1$ and the other that lies in the bottom row $j + 1$. For generating the solution in the column $i + 1$, we need to check if the solution above it (row $j - 1$) is already removed from the min-heap (lines 17-19). For this, the $(i+1)^{th}$ entry in $flag$ array is inspected. Similarly, to generate the solution in the row $j + 1$, we need to check if the solution left to it (column $i - 1$) is already removed from the min-heap (lines 20-23). For this, the $(i - 1)^{th}$ entry in $flag$ array is inspected. This entire process repeats until all $r \cdot t$ solutions are generated. In the end, the stack $N$ contains the set of non-dominated solutions to the combined problem which is returned to the calling function (line 24).

Theorem 4. *The memory requirement of the CombineEff procedure is $O(r) + |N|$, where $r = |E_1|$ and $N \subseteq E$ is the set of non-dominated solutions to the combined problem.*

We claim that the size of the min-heap $H$ at any point in time during execution never exceeds $r$. Suppose that the candidate solutions $E$ to the problem are arranged in $t \times r$ table as depicted in (Figure 1c). If there are more than $r$ tuples in min-heap, then by pigeonhole principle at least two tuples, say $e_u$ and $e_v$ must belong to the same column. Suppose that $u$ and $v$ indicate the row numbers of these tuples and let $u < v$. Then by Theorem 3, $e_u$ must be smaller than $e_v$ with respect to the first objective $f_1$, i.e., $e_u.f_1 < e_v.f_1$ ($e_u.f_1$ and $e_v.f_1$ represents the first objective value of the tuples $e_u$ and $e_v$, respectively). However, $CombineEff$ does not generate $e_v$ until $e_u$ is removed from the min-heap. Therefore, tuples $e_u$ and $e_v$ cannot not be present in the min-heap at the same time. Hence, the contradiction.

The size of the integer array $flag$ is also $O(r)$. Further, the stack $N$ only stores the non-dominated solutions. Therefore, the total memory requirement of the $CombineEff$ procedure is $O(r) + |N|$, where $N \subseteq E$ is the set of non-dominated solutions to the combined problem. As the *for* loop runs for $r \cdot t$ iterations and the maximum amount of work done in each loop is $O(log(r))$ (min-heap operations), the time complexity of the $CombineEff$ procedure is $O(r \cdot t \cdot log(r))$. Note: Algorithm 2 assumes that solutions in both sets $E_1$ and $E_2$ are sorted by the first objective. It generates all possible $r \cdot t$ solutions in a strategic manner. The same algorithm works well (with minimum modifications) even if we sort solutions by the second objective.

**Illustration**: Consider two sets of non-dominated solutions $E_1$ and $E_2$ (sorted by the first objective) given in Table 2. The combined set of solutions $E$ is also shown in Table 2. The non-dominated solutions are highlighted in bold. Table 3 illustrates the contents of

**Algorithm 2** Efficient Algorithm to Combine Non-dominated Solutions of Two Sub-problems

1: **procedure** $CombineEff$
2: **Input:** Two sets of non-dominated solutions $E_1$ and $E_2$. Both sets are sorted by the first objective.
3: **Output:** A set of non-dominated solutions $N$ in the combined set $E = \{(a+c, b+d) | (a,b) \in E_1 \; and \; (c,d) \in E_2\}$
4: initialize stack $N$
5: $N.push(0, \infty)$
6: initialize min-heap $H$ of size $r$
7: $H.add((a_1 + c_1, b_1 + d_1))$
8: $flag[\,] = int[r]$
9: **for** $l = 1$ to $r \cdot t$ **do**
10: $(f_1, f_2) = H.remove()$ //$min$ solution
11: **if** $f_2 < N.top().f_2$ **then**
12: $N.push(f_1, f_2)$ //increments top and pushes element into stack
13: **end if**
14: $i$ = col index of $min$ tuple $(f_1, f_2)$
15: $j$ = row index of $min$ tuple $(f_1, f_2)$
16: $flag[i] = j$ //update entry in pos $i$ with the row index $j$ of $min$ soln
17: **if** $i + 1 \leq r$ and $(j == 1$ or $flag[i+1] == j - 1)$ **then**
18: $H.add((a_{i+1} + c_j, b_{i+1}, d_j))$ // generate right tuple
19: **end if**
20: **if** $j + 1 \leq t$ and $(i == 1$ or $flag[i-1] \geq j + 1)$ **then**
21: $H.add((a_i + c_{j+1}, b_i + d_{j+1}))$ //generate bottom tuple
22: **end if**
23: **end for**
24: return solutions in stack $N$
25: **end procedure**

**Table 2: Two sets $E_1$ and $E_2$ of non-dominated solutions each containing four tuples (sorted by the first objective), and their combination $E$ consisting of $4 \cdot 4 = 16$ candidate solutions. Non-dominated solutions are highlighted in bold.**

| $E_1$ | $E_2$ | Combined E | | | |
|---|---|---|---|---|---|
| (1,4) | (2,10) | **(3,14)** | (7,13) | (8,12) | (15,11) |
| (5,3) | (4,7) | **(5,11)** | (9,10) | (10,9) | (17,8) |
| (6,2) | (5,6) | **(6,10)** | (10,9) | **(11,8)** | (18,7) |
| (13,1) | (7,5) | **(8,9)** | (12,8) | **(13,7)** | **(20,6)** |

heap $H$, stack $N$ and array $flag$ during the iterations of Algorithm 2. In the beginning, a dummy entry $(0, \infty)$ is added to the stack (line 5). Further, the solution tuple (1,4) from $E_1$ and the solution tuple (2,10) from $E_2$ are combined to produce (3,14) and added to the heap (line 7). Thus, at the start of the first iteration, that is, $l = 1$, heap $H$ contains (3,14). Subsequently, it is removed from the heap and compared with the top element in stack $N$. As the second objective value of the top element is $\infty$, the tuple (3,14) is added to the stack (lines 11-13) and the dummy entry $(0, \infty)$ is removed. As the first entry in the first column is generated and removed from the heap, $flag[0]$ is set to 1. Subsequently, the solution tuple (5,3) from $E_1$ and the solution tuple (2,10) from $E_2$ are combined to obtain (7,13), and added to the heap (lines 17-19). Similarly, the solution tuple (1,4) from $E_1$ and the solution tuple (4,7) from $E_2$ are combined to obtain (5,11), and added to the heap (lines 20-22). Thus, at the start of the second iteration, that is, $l = 2$, heap $H$ contains (5,11) and (7,13), and stack $N$ contains a non-dominated solution (3,14). In this way, in each iteration, the algorithm removes a tuple from heap $H$, pushes it on stack $N$ if appropriate, and generates at most two new tuples and adds them to heap $H$.

**Table 3: Execution of Algorithm 2 on solutions $E_1$ and $E_2$ (Table 2) of two sub-problems (sorted by the first objective).**

| Iter $l$ | Heap $H$ | Stack $N$ | $flag$ |
|---|---|---|---|
| 1 | (3,14) | $(0, \infty)$ | [0,0,0,0] |
| 2 | (5,11)(7,13) | (3,14) | [1,0,0,0] |
| 3 | (6,10)(7,13) | (3,14)(5,11) | [2,0,0,0] |
| 4 | (7,13)(8,9) | (3,14)(5,11)(6,10) | [3,0,0,0] |
| 5 | (8,9)(9,10)(8,12) | (3,14)(5,11)(6,10) | [3,1,0,0] |
| 6 | (8,12)(9,10) | (3,14)(5,11)(6,10)(8,9) | [4,1,0,0] |
| 7 | (9,10)(15,11) | (3,14)(5,11)(6,10)(8,9) | [4,1,1,0] |
| 8 | (10,9)(15,11)(10,9) | (3,14)(5,11)(6,10)(8,9) | [4,2,1,0] |
| 9 | (10,9)(15,11) | (3,14)(5,11)(6,10)(8,9) | [4,2,2,0] |
| 10 | (11,8)(15,11)(12,8) | (3,14)(5,11)(6,10)(8,9) | [4,3,2,0] |
| 11 | (12,8)(15,11) | (3,14)…(8,9)(11,8) | [4,3,3,0] |
| 12 | (13,7)(15,11) | (3,14)…(8,9)(11,8) | [4,4,3,0] |
| 13 | (15,11) | (3,14)…(11,8)(13,7) | [4,4,4,0] |
| 14 | (17,8) | (3,14)…(11,8)(13,7) | [4,4,4,1] |
| 15 | (18,7) | (3,14)…(11,8)(13,7) | [4,4,4,2] |
| 16 | (20,6) | (3,14)…(11,8)(13,7) | [4,4,4,3] |
| 17 | | (3,14)…(13,7)(20,6) | [4,4,4,4] |

## 3.2 Evolutionary Algorithms

Classical optimization methods typically convert a multi-objective problem to a single objective problem by assigning appropriate weights to each objective. Such methods yield only one Pareto-optimal solution for the given weights. To obtain multiple solutions, they have to be applied many times. On the other hand, evolutionary algorithms are considered to be more effective in solving multi-objective optimization problems since they work with a population of solutions and produce a set of non-dominated solutions in a single simulation run. Several multi-objective evolutionary algorithms have been proposed in the literature. NSGAII is an extended genetic algorithm that uses the concept of elitism, fast non-dominated sorting and crowding distance (for diversity) to solve multi-objective problems [12]. Recently, a binary particle swarm optimizer BMOPSOCD was proposed to solve WSLAP [3] and was shown to generate better solutions than NSGAII. BMOPSOCD maintains a set of non-dominated solutions in the external archive and uses crowding distance to find diverse solutions. In this work, we make use of NSGAII and BMOPSOCD to find the non-dominated solutions for each web service in WSLAP.

# 4 EXPERIMENT

We compare the efficacy of the proposed D&C approach with multi-objective evolutionary algorithms (MOEAs) proposed in the literature [3, 6, 7] on test instances of different sizes. Specifically, we consider four algorithms: NSGAII, BMOPSOCD, D&C NSGAII and D&C BMOPSOCD. All algorithms were implemented in Python version 3.7 and the experiments were conducted on a Windows 10 machine with i7-8650U 2.11 GHz processor and 16GB of RAM. We follow the experimental methodology used by [3].

## 4.1 Test Instances

We conducted experiments with 14 different WSLAP instances described in [3]. The search space of an instance is computed using the number of services and candidate locations ($2^{s \cdot n}$) as shown in Table 4. Since the computation complexity of an instance also depends on the number of user centers, for each search space, the number of user centers is varied. The test instances employ real-world WSDream dataset [13] for obtaining latency numbers. This dataset contains only latencies between candidate locations and user centers, and lacks deployment costs for candidate locations and invocation frequencies for web services. Therefore, to generate a complete test instance, we followed the approach suggested in [3]. We randomly generated the deployment costs for candidate locations according to a normal distribution with a mean of 100 and a standard deviation of 20. We also randomly generated the invocation frequencies for user centers from a uniform distribution between 1 and 120.

**Table 4: Different WSLAP Instances Used in Experiments.**

| Instance | Services $s$ | Locations $m$ | User Centers $n$ | Search Space $2^{s \cdot n}$ |
|---|---|---|---|---|
| 1 | 20 | 5 | 10 | $2^{100}$ |
| 2 | 20 | 10 | 10 | $2^{200}$ |
| 3 | 50 | 15 | 20 | $2^{750}$ |
| 4 | 50 | 15 | 40 | $2^{750}$ |
| 5 | 50 | 25 | 20 | $2^{1250}$ |
| 6 | 50 | 25 | 40 | $2^{1250}$ |
| 7 | 100 | 15 | 20 | $2^{1500}$ |
| 8 | 100 | 15 | 40 | $2^{1500}$ |
| 9 | 100 | 25 | 20 | $2^{2500}$ |
| 10 | 100 | 25 | 40 | $2^{2500}$ |
| 11 | 200 | 25 | 40 | $2^{5000}$ |
| 12 | 200 | 25 | 80 | $2^{5000}$ |
| 13 | 200 | 40 | 40 | $2^{8000}$ |
| 14 | 200 | 40 | 80 | $2^{8000}$ |

## 4.2 Performance Metrics

We use HyperVolume (HV) [14, 15] and Inverted Generational Distance (IGD) [16] to evaluate the diversity and quality of solutions produced by four algorithms (NSGAII, BMOPSOCD, D&C NSGAII and D&C BMOPSOCD). HV is a measure that reflects the volume enclosed by a solution set and a reference point. A larger HV value indicates a better solution set. IGD is a modified version of Generational Distance (GD) [17] which estimates how far the elements in the true Pareto front are from those in the non-dominated set produced by an algorithm. IGD calculates the sum of the distances from each point within the true Pareto front to the nearest point within the non-dominated set produced by an algorithm. A lower IGD value indicates a better quality solution set. However, for calculating the IGD value, we need a true Pareto front, and for our problem, the true Pareto front is unknown. Therefore, as discussed in [3], we computed an approximated Pareto front by combining all solutions produced by four algorithms and then applying a non-dominated sorting to obtain the final non-dominated set.

## 4.3 Parameter Settings

The parameter values used for each algorithm considered in the evaluation are shown in Table 5. We found the parameter values for NSGAII algorithm empirically, that is, we tried several values for parameters, and observed whether the solutions have converged (have similar fitness values between two consecutive generations). We used the same values of crossover probability (0.8) and mutation probability (0.2) for D&C NSGAII. The size of a chromosome used for NSGAII is $s \cdot n$, where $s$ is the number of web services and $n$ is the number of candidate locations. Since D&C NSGAII, solves the location allocation for each web service independently, the size of a chromosome, in this case, is just $n$. As the search space of NSGAII is larger $2^{s \cdot n}$, we used a population size of 250. As the search space of D&C NSGAII is smaller $2^n$, we used a smaller population of size 20. Note that, our study focuses on the effectiveness of D&C approach instead of selecting the best parameter set.

For BMOPSOCD algorithm, we used the parameter values as described in [3]. The value of static inertia weight $w$ was set to 0.4 and the mutation probability $P_m$ was set to 0.5. The parameters $c_1$ and $c_2$ were both set to 1. Hence, particle's personal best and swarm's global best had an equal influence on the swarm.

**Table 5: Parameters Choice for the Four Algorithms.**

| *Parameter* | *NSGAII* | *D&C NSGAII* |
|---|---|---|
| Population size | 250 | 20 |
| Chromosome size | $s \cdot n$ | $n$ |
| Tournament size | 3 | 3 |
| Crossover probability | 0.8 | 0.8 |
| Mutation probability | 0.2 | 0.2 |
| Maximum generations | 250 | 40 |
| *Parameter* | *BMOPSOCD* | *D&C BMOPSOCD* |
| Population size | 250 | 20 |
| Archive size | 250 | 20 |
| Inertia $w$ | 0.4 | 0.4 |
| Personal best $c1$ | 1 | 1 |
| Swarm best $c2$ | 1 | 1 |
| Mutation probability $p_m$ | 0.5 | 0.5 |
| Maximum generations | 250 | 40 |

We normalized two objective functions (latency and cost) between 0 and 1. The point (1, 1) is the extreme point of objective values. We used (1, 1) as the reference point in calculating HV. For each experiment, the proposed algorithm was run ten times independently. The best results of all the runs were compared. To obtain the best result of ten runs, the results of all ten runs were combined and sorted by the non-dominated values.

## 4.4 Results

Figure 2a depicts the set of non-dominated solutions produced by the four algorithms for one of the instances (Instance 3). For better clarity, we have shown the closer view of the non-dominated solutions obtained for the instance in figure 2b. The figure clearly shows that D&C NSGAII and D&C BMOPSOCD are able to find better and diverse solutions than NSGAII and BMOPSOCD. Similar trends were observed for other instances. Although NSGA-II and

**Table 6: HV Values from the Four Algorithms**

| Inst--ance | NSGA II | | D&C NSGA II | | BMOPSOCD | | D&C BMOPSOCD | |
|---|---|---|---|---|---|---|---|---|
| | Mean | Std | Mean | Std | Mean | Std | Mean | Std |
| 1 | 0.72 | 0.03 | 0.82 | 0.01 | 0.78 | 0.01 | 0.90 | 0.00 |
| 2 | 0.68 | 0.07 | 0.92 | 0.00 | 0.89 | 0.01 | 0.96 | 0.00 |
| 3 | 0.55 | 0.09 | 0.95 | 0.00 | 0.89 | 0.01 | 0.97 | 0.00 |
| 4 | 0.56 | 0.09 | 0.95 | 0.00 | 0.90 | 0.01 | 0.97 | 0.00 |
| 5 | 0.54 | 0.10 | 0.97 | 0.00 | 0.93 | 0.01 | 0.98 | 0.00 |
| 6 | 0.55 | 0.10 | 0.97 | 0.00 | 0.93 | 0.00 | 0.98 | 0.00 |
| 7 | 0.54 | 0.09 | 0.95 | 0.00 | 0.86 | 0.01 | 0.97 | 0.00 |
| 8 | 0.55 | 0.09 | 0.95 | 0.00 | 0.87 | 0.01 | 0.97 | 0.00 |
| 9 | 0.54 | 0.10 | 0.97 | 0.00 | 0.90 | 0.01 | 0.98 | 0.00 |
| 10 | 0.54 | 0.10 | 0.96 | 0.00 | 0.91 | 0.01 | 0.98 | 0.00 |
| 11 | 0.51 | 0.00 | 0.96 | 0.00 | 0.88 | 0.01 | 0.98 | 0.00 |
| 12 | 0.51 | 0.00 | 0.96 | 0.00 | 0.89 | 0.01 | 0.98 | 0.00 |
| 13 | 0.51 | 0.00 | 0.95 | 0.00 | 0.92 | 0.00 | 0.98 | 0.00 |
| 14 | 0.51 | 0.00 | 0.94 | 0.00 | 0.92 | 0.00 | 0.98 | 0.00 |

**Table 7: IGD Values from the Four Algorithms**

| NSGA II | | D&C NSGA II | | BMOPSOCD | | D&C BMOPSOCD | |
|---|---|---|---|---|---|---|---|
| Mean | Std | Mean | Std | Mean | Std | Mean | Std |
| 0.19 | 0.02 | 0.14 | 0.00 | 0.15 | 0.01 | 0.00 | 0.00 |
| 0.17 | 0.04 | 0.07 | 0.00 | 0.07 | 0.00 | 0.00 | 0.00 |
| 0.34 | 0.07 | 0.05 | 0.01 | 0.09 | 0.01 | 0.00 | 0.00 |
| 0.25 | 0.05 | 0.04 | 0.01 | 0.05 | 0.01 | 0.00 | 0.00 |
| 0.32 | 0.07 | 0.03 | 0.01 | 0.04 | 0.01 | 0.00 | 0.00 |
| 0.30 | 0.07 | 0.03 | 0.00 | 0.04 | 0.01 | 0.00 | 0.00 |
| 0.39 | 0.07 | 0.08 | 0.01 | 0.13 | 0.01 | 0.00 | 0.00 |
| 0.26 | 0.05 | 0.04 | 0.01 | 0.07 | 0.01 | 0.00 | 0.00 |
| 0.28 | 0.06 | 0.03 | 0.00 | 0.04 | 0.00 | 0.00 | 0.00 |
| 0.30 | 0.06 | 0.04 | 0.01 | 0.05 | 0.01 | 0.00 | 0.00 |
| 0.27 | 0.01 | 0.04 | 0.01 | 0.05 | 0.01 | 0.00 | 0.00 |
| 0.26 | 0.01 | 0.04 | 0.00 | 0.05 | 0.00 | 0.00 | 0.00 |
| 0.30 | 0.01 | 0.04 | 0.01 | 0.04 | 0.01 | 0.00 | 0.00 |
| 0.28 | 0.01 | 0.04 | 0.01 | 0.03 | 0.01 | 0.00 | 0.00 |

BMOPSOCD have longer tails (i.e., many solutions with cost > 0.5) than D&C NSGAII and D&C BMOPSOCD, the solutions present in the tails are dominated by the solutions of the D&C algorithms. Table 6 shows HV values and Table 7 shows IGD values (all values are rounded to two decimal places) calculated using non-dominated solutions obtained by each algorithm for all fourteen instances. A larger HV value indicates a better and diverse solution set. A lower IGD value indicates a better quality solution set. From the Tables 6 and 7, it is clear that solutions obtained using the D&C approach are much better than the ones obtained using the combined approach in terms of quality as well as diversity. Overall, D&C BMOPSOCD produced better solutions than the other three algorithms. Further, as observed in [3], BMOPSOCD produced better results than NSGAII. However, D&C NSGAII resulted in better solutions than BMOPSOCD.

**Table 8: Computation Time (in sec) of the Four Algorithms**

| Inst--ance | NSGA II | | D&C NSGA II | | BMOPSOCD | | D&C BMOPSOCD | |
|---|---|---|---|---|---|---|---|---|
| | Mean | Std | Mean | Std | Mean | Std | Mean | Std |
| 1 | 22 | 1 | 9 | 2 | 13 | 1 | 6 | 2 |
| 2 | 41 | 1 | 12 | 1 | 21 | 2 | 6 | 1 |
| 3 | 283 | 8 | 53 | 5 | 108 | 6 | 24 | 3 |
| 4 | 577 | 45 | 99 | 14 | 192 | 7 | 40 | 6 |
| 5 | 446 | 16 | 98 | 16 | 194 | 6 | 42 | 8 |
| 6 | 922 | 96 | 192 | 36 | 318 | 19 | 55 | 6 |
| 7 | 566 | 29 | 90 | 6 | 264 | 13 | 69 | 9 |
| 8 | 1018 | 30 | 216 | 16 | 417 | 15 | 144 | 9 |
| 9 | 1103 | 95 | 300 | 26 | 278 | 17 | 167 | 12 |
| 10 | 1913 | 88 | 480 | 63 | 404 | 7 | 175 | 12 |
| 11 | 909 | 95 | 1982 | 159 | 847 | 21 | 829 | 56 |
| 12 | 1775 | 295 | 2843 | 258 | 1291 | 16 | 936 | 43 |
| 13 | 1227 | 119 | 1887 | 213 | 1478 | 368 | 906 | 168 |
| 14 | 2499 | 187 | 2880 | 270 | 5031 | 413 | 998 | 60 |

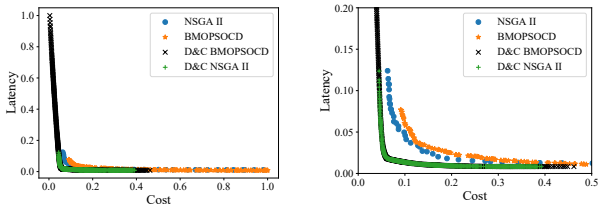


**Figure 2: a) Pareto fronts obtained from the four algorithms for instance 3 (50,15,20). b) Closer view of the pareto fronts.**

## 5 CONCLUSION AND FUTURE WORK

In this paper, we show that WSLAP can be solved effectively using a Divide and Conquer (D&C) approach; wherein, the location-allocation problem is solved independently for each web service. Further, we propose a novel merge algorithm to combined the solution from each service. We used NSGA-II and BMOPSOCD to compute the Pareto front for each service. We compared performance of both the algorithm with and without D&C approach. Our extensive experiments showed that the D&C approach is able to produce better quality (lower Inverted Generational Distance) and diverse solutions (higher HyperVolume) than solving the WSLAP as one problem. The computational time is also less for D&C approach.

Further, the computational time can be significantly reduced by exploiting the parallelization capability of the proposed algorithm to merge the non-dominated solutions of multiple services simultaneously. Specifically, we can generate solutions for each service in parallel, and, during merge, we could do pairwise merges in parallel going all the way up the complete binary tree. Since the proposed merge algorithm is generic, we plan to study the effectiveness of the algorithm for other bi-objective optimization problems.

# REFERENCES


[1] D. A. Menasce. QoS Issues in Web Services. *IEEE Internet Computing*, 6(6):72–75, Nov 2002.

[2] Arun Ramamurthy, Saket Saurabh, Mangesh Gharote, and Sachin Lodha. Selection of cloud service providers for hosting web applications in a multi-cloud environment. In *2020 IEEE International Conference on Services Computing (SCC)*, pages 202–209. IEEE, 2020.

[3] Y. Mei B. Tan, H. Ma and M. Zhang. Evolutionary Multi-Objective Optimization for Web Service Location Allocation Problem. *IEEE Transactions on Services Computing*, pages 1–1, 2018.

[4] Reza Zanjirani Farahani and Masoud Hekmatfar. *Facility Location: Concepts, Models, Algorithms and Case studies*. Springer, 2009.

[5] Reza Zanjirani Farahani, Maryam SteadieSeifi, and Nasrin Asgari. Multiple criteria facility location problems: A survey. *Applied Mathematical Modelling*, 34(7):1689 – 1709, 2010.

[6] Boxiong Tan, Yi Mei, Hui Ma, and Mengjie Zhang. Particle swarm optimization for multi-objective web service location allocation. In Francisco Chicano, Bin Hu, and Pablo García-Sánchez, editors, *Evolutionary Computation in Combinatorial Optimization*, pages 219–234, Cham, 2016. Springer International Publishing.

[7] Boxiong Tan, Hui Ma, and Mengjie Zhang. Optimization of Location Allocation of Web Services Using a Modified Non-dominated Sorting Genetic Algorithm. In *Proceedings of the Second Australasian Conference on Artificial Life and Computational Intelligence - Volume 9592*, pages 246–257, Berlin, Heidelberg, 2016. Springer-Verlag.

[8] Hui Ma, Alexandre Sawczuk da Silva, and Wentao Kuang. Nsga-ii with local search for multi-objective application deployment in multi-cloud. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 2800–2807. IEEE, 2019.

[9] Tao Shi, Hui Ma, and Gang Chen. Divide and conquer: Seeding strategies for multi-objective multi-cloud composite applications deployment. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 317–318, 2020.

[10] H. T. Kung, F. Luccio, and F. P. Preparata. On Finding the Maxima of a Set of Vectors. *J. ACM*, 22(4):469–476, October 1975.

[11] Carlo A Furia, Bertrand Meyer, and Sergey Velder. Loop invariants: Analysis, classification, and examples. *ACM Computing Surveys (CSUR)*, 46(3):1–51, 2014.

[12] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[13] Y. Zhang, Z. Zheng, and M. R. Lyu. Exploring Latent Features for Memory-Based QoS Prediction in Cloud Computing. In *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, pages 1–10, 2011.

[14] 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.

[15] N. Riquelme, C. Von Lücken, and B. Baran. Performance metrics in multi-objective optimization. In *2015 Latin American Computing Conference (CLEI)*, pages 1–11, 2015.

[16] Carlos A Coello Coello and Margarita Reyes Sierra. A Study of the Parallelization of a Coevolutionary Multi-objective Evolutionary Algorithm. In *Mexican international conference on artificial intelligence*, pages 688–697. Springer, 2004.

[17] D. A. Van Veldhuizen and G. B. Lamont. On Measuring Multiobjective Evolutionary Algorithm Performance. In *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*, volume 1, pages 204–211 vol.1, 2000.