Community-Oriented Resource Allocation at the Extreme Edge

Abdalla A. Moustafa*, Sara A. Elsayed†, Hossam S. Hassanein†
* Department of Electrical and Computer Engineering, Queen's University, Kingston, ON, Canada
† School of Computing, Queen's University, Kingston, ON, Canada
abdalla.moustafa@queensu.ca, selsayed@cs.queensu.ca, hossam@cs.queensu.ca

GLOBECOM 2022 - 2022 IEEE Global Communications Conference | 978-1-6654-3540-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/GLOBECOM48099.2022.10001198

***Abstract*—The surging demand for Edge Computing (EC) to cope with the proliferation of latency-critical and data-intensive applications has inspired the notion of recycling ample yet underutilized computational resources of end devices, also referred to as Extreme Edge Devices (EEDs). Maintaining data privacy and cost efficiency remain core challenges for the viability of EED-enabled computing paradigms. In this context, we propose the Community-Oriented Resource Allocation (CORA) scheme. CORA exploits business, institutional, and social relationships to build clusters and communities of requesters and EEDs that can eliminate recruitment costs and preserve privacy. However, community-imposed constraints on resource allocation can lead to unbalanced work distribution. To address this issue, CORA considers community restrictions, minimizes flowtime and makespan for the allocated services, and retains a reasonable scheduler runtime for real-time resource allocation. Towards that end, CORA formulates the resource allocation problem as a Bipartite Graph Matching problem. Furthermore, CORA exposes tuneable parameters that allow prioritizing flowtime or makespan, making it suitable for different scenarios. Extensive simulations show that CORA outperforms six prominent heuristic-based resource allocation schemes by up to 24% in terms of average makespan while sustaining the same level of flowtime and runtime.**

***Index Terms*—Edge Computing, Extreme Edge, EEDs, Resource Allocation, Container Placement, Graph Processing.**

I. Introduction

With the progressively adopted vision of the Internet of Things (IoT), it is foreseen that 125 billion IoT devices will be connected to the Internet by 2030 [1]. This proliferation is expected to increase the momentum of IoT applications and services that require heavy processing and stringent Quality of Service (QoS), including machine learning, augmented reality, tactile internet, and healthcare applications [2]. Cloud Computing (CC) fails to accommodate the severe QoS requirements of such applications, since it requires full transmission of excessive amount of data to far-away data centers, which can significantly increase latency and inflict huge traffic influx at backhaul links [3].

Edge Computing (EC) is a promising paradigm that can resolve the aforementioned issues by granting the computing service closer to end-users [4]. However, the dominant majority of existing EC platforms and models fall solely under the control of cloud service providers and/or network operators [5]. Challenging this monopoly by recycling ample yet underutilized computational resources of Extreme Edge Devices (EEDs) can democratize the edge and open a new market for more players to manufacture and administer their own edge cloud. This market can enable individuals, businesses, enterprises, and even municipalities to act as edge service providers themselves and/or monetize their computing resources. In addition, EED-enabled computing paradigms can bring the computing service much closer to end-users, drastically diminishing the delay [6].

Despite its advantageous impact, EED-enabled computing is considered less secure than infrastructure-based EC paradigms. This is due to relying on dubious machines. In addition, the need to recruit many EEDs for parallel execution of a single partitioned task can result in significant recruitment costs. HomeEdge [7] is an EED-enabled computing platform that addresses these problems by offloading tasks to other devices in the local network owned by the same user. This can ensure a higher level of privacy since all devices and applications that run on those devices are trusted by the user. Moreover, latency can be drastically reduced because of the proximity factor. Finally, users do not pay to use their own devices. However, restricting the scope to the local network severely limits the resource pool, which in turn reduces the utilization gains and the chance of finding a suitable device for task offloading. This paper avoids such restriction by proposing the Community-Oriented Resource Allocation (CORA) scheme.

CORA leverages the underutilized computational resources of EEDs and exploits the broad sense of community. Such a community can be a neighborhood, a group of friends, a hospital, or devices owned by an organization in different geographic locations worldwide with different time zones or load peak times. In other words, CORA fosters the concept of service for service exchange. The goal is to create a global network where users are allowed to form separate communities of trusted users, each owning one or more devices. This significantly expands the scope of the available pool of resources while preserving privacy and eliminating any recruitment cost. In addition to communities, CORA enables the creation of clusters of devices in close proximity to each other. This enables each device to prioritize offloading its tasks to other devices in its cluster for lower latency and even higher security while having the fallback option of offloading to devices anywhere else in the world as long as they are included in one of the user's communities.

Despite its benefits, the notion of community adds another

978-1-6654-3540-6/22/$31.00

dimension to resource allocation that can cause problems for schemes that ignore the restrictions imposed by communities in terms of the order of assignment. In this context, CORA proposes a new resource allocation scheme using a graph-directed approach to allocate container-based services in community-oriented EED-enabled edge computing environments. To the best of our knowledge, CORA is the first community-based resource allocation scheme that overcomes the constraints introduced by communities. We evaluate the performance of CORA compared to six other prominent resource allocation approaches in container-based schedulers. Extensive simulations have shown that CORA yields significant improvements, reaching up to 24% in terms of the average makespan while sustaining the same level of flowtime and runtime.

The remainder of this paper is organized as follows. Section II overviews some of the related work. Section III introduces our scheme (CORA). Section IV reports the performance evaluation and simulation results. Section V concludes the findings and discusses future work.

II. Related Work

The desire to give users the freedom of executing a wide range of services has led to the choice of container-based services. Recently, container allocation research has seen a surge in popularity due to its viability as an offloading mechanism [8]. Since container allocation is an NP-hard problem. The resource allocation techniques for containers can be divided into four major categories; mathematical modeling, heuristics, meta-heuristics, and machine learning [8].

Mathematical approaches model the resource allocation problem as a set of constrained equations and then try to solve them. Approaches under this category suffer from being NP-complete in complexity [8]. Thus, large-scale problems cannot be solved in polynomial time. Zhou et al. [9] formulate the resource allocation problem as an optimization program that minimizes certain function values while considering tasks deadline and resource constraints. Other schemes are optimized for work cost in terms of energy consumption [10].

Multiple heuristics have been proposed to allocate containers [11], [12]. These approaches are generally faster and more scalable than the approaches mentioned above. However, the solutions are not guaranteed to be optimal. One of the simplest yet most prominent heuristics is the work queue, which is used in Google Kubernetes [13]. The work queue selects a task randomly and assigns it to the device with the minimum workload and/or maximum available resources. Several other common resource allocation techniques that can be forced to follow community constraints. Min-min [11] uses the minimum completion time as a metric, meaning that the task that can be completed the earliest is given a higher priority. Max-min [11] starts like the Min-min by calculating the minimum completion time for every service. However, then it proceeds to select the one rendering the maximum-minimum completion time. LJFR SJFR [12] is a combination of both the Min-min and Max-min heuristics, as it alternates between them by assigning the longest service to the fastest available device, then the shortest service to the fastest available device. This sequence repeats until all jobs has been assigned. In the Sufferage heuristic [12], priority is given to the service that suffers the most from not assigning it at the current step. This is done by calculating the difference between the minimum and second minimum completion times for every service and choosing the one with the maximum sufferage.

To the best of our knowledge, none of the existing resource allocation schemes is optimized for communities. Some schemes address the concept of matching constraints for allocation. For example, in [14], the authors recommend an online resource allocation scheme with matching constraints that optimize for time-changing cost. However, the scheme assumes that the cost function is known at any point in time. In addition, it is based on two linear programming algorithms that suffer from poor scalability. Such approaches still fail to address the restrictions associated with communities. In contrast to existing schemes, we propose a resource allocation scheme optimized for communities.

III. Community-Oriented Resource Allocation (CORA)

A. System Model and Overview

In CORA, a device can be any machine, stationary or portable. Any device running the software can act as a requester and/or a worker. Note that we interchangeably use the terms EEDs and workers throughout the paper. Each cluster is composed of the collection of devices that are within the local network owned by the same user, with a single device acting as the local scheduler for this network. A communication gate exists between the cluster and the edge server that connects it to other clusters. A community is composed of one or more clusters. A community refers to a group of users that are open to exchanging services and executing offloaded tasks among each other. Note that a user can be a member of multiple communities.

To further illustrate the concept of clusters and communities in CORA, Fig. 1 depicts a system of 4 clusters owned by a total of three users (user A, user B, and user C), which form two communities (community X and community Y). Any device that is a member of a cluster owned by user B can offload services to any device owned by users A or B. This is since user A and user B are both members of community X. In contrast, the devices of user A can exchange services with users B and C alike. These constraints explain the need for a community-oriented scheduler. For example, consider the case where a generic scheduler allocates service 2 to worker A because of a slight time advantage. Now, the same scheduler is forced to allocate service 1 to worker A as well because it cannot be allocated to worker B. This can lead to stacking multiple services on one device while leaving other available devices unutilized.

CORA enables users to offload their custom tasks, with as few limitations as possible, to trusted devices that are members of their own community. This is done regardless of whether the devices are within the same cluster or in a remote location, or

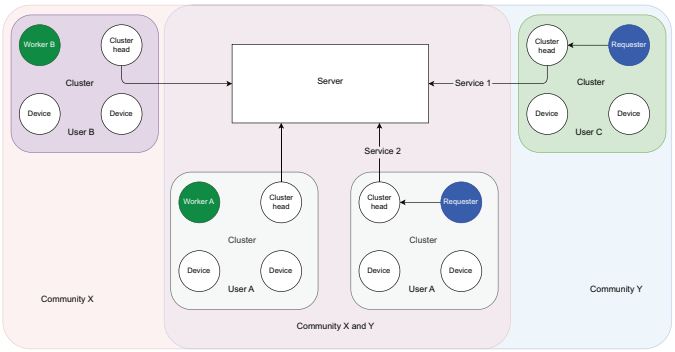

Fig. 1. Clusters, Users, and Communities Relation.

whether all clusters in this community are owned by a single user, a single organization, or a collection of entities, as long as all cluster owners trust their community members. In this system, all connected devices can act as requesters, workers, or cluster heads depending on the configuration and the running scenario. We start on a cluster level, where every collection of devices on a LAN can be handled by a single device known as the cluster head. These cluster head, in turn, have access to the server scheduler that can receive and schedule services between different clusters within the same community.

Fostering user custom applications in CEP requires overcoming some new challenges, especially when dealing with a wide range of devices and operating systems. Therefore, we opt to use the containers ecosystem. This ecosystem allows the requester to provide almost any source code as a task that can be offloaded in a containerized form that can run on a large, diverse group of smart devices, with no regard to the programming language, required libraries, or application domain. In addition to flexibility on the software side, this enables the services to run on any Docker Container-enabled devices. Docker containers are selected due to their quick deployment, easy management, safety, and hardware independence [8].

In CORA, each device is a member of a cluster that belongs to a specific user, who in turn is a member of one or more communities. Let $S = \{s_1, s_2, \ldots, s_n\}$ denote the set of containerized services in the server queue to be offloaded to edge devices within communities. We assume independent services (with no inter-service data dependencies), and preemption is not allowed since the container state cannot be relocated to a new device without wasting additional resources. Each service originates from a requester within a cluster that belongs to a user with a set of valid communities, each of which has candidate workers to which the service can be offloaded safely. Moreover, the set of workers available for service assignment is denoted $D = \{d_1, d_2, \ldots, d_m\}$. Each worker d_j has its updated benchmarks, which indicate the available resources, as well as a cluster identifier. The latter can be traced back to the corresponding user and thus to the associated set of communities. On the machine level, the assigned services are executed in a First-Come, First-Served (FCFS) order.

B. CORA at the Scheduler

By checking for union values between the data from a service community set and a worker set of communities, we can infer the set of workers that are eligible to execute each service. This problem can be better represented by a graph of vertices as given by equation (1), which is the union of the workers and services sets. This graph can be divided into two groups, one for services and one for workers. The vertices from the first group (i.e., services) can only have an edge with vertices from the second group (i.e., workers). Thus, the graph can be defined as a bipartite graph.

$$\begin{aligned} V &= S \cup D \\ &= \{v_1, v_2, \ldots, v_n, v_{n+1}, v_{n+2}, \ldots, v_{n+m}\} \end{aligned} \tag{1}$$

In the previously mentioned bipartite graph, an edge between two vertices exists only if they share at least one community, as reflected by equation (2), where the Expected Time to Compute (ETC) is an $n \times m$ matrix in which n is the number of services and m is the number of workers. Each entry in the ETC matrix represents the estimated execution time for a given service on each worker. Thus, for each service s_i and each worker d_j, $ETC(s_i, d_j)$ is the estimated execution time of s_i on d_j. We define C_i as the set of communities for service s_i and C_j as the community set for d_j. Each edge in the graph connects one service to one worker, where the intersection between C_i and C_j is not empty, and the edge weight is the corresponding value from the ETC matrix.

$$\begin{aligned} e_{i,j} &= \text{ETC}(i,j) \\ E_{\text{all}} &= \{e_{1,1}, e_{1,2}, \ldots, e_{1,m}, e_{2,1}, \ldots, e_{n,m}\} \\ E &= \{e_{i,j} \in E_{\text{all}} | C_i \cap C_j \neq \Phi\} \end{aligned} \tag{2}$$

The approximate time it takes worker d_j to complete all assigned services is denoted t_j, and is given by equation (3), where W_j represents the prior workload on the worker, and A_j indicates the set of services assigned to this worker by the scheduler.

$$t_j = \sum_{i \in A_j} e_{i,j} + W_j \tag{3}$$

CORA strives to minimize the average makespan and flowtime. Flowtime is the sum of the execution time of all services on their selected workers, as given by equation (4). Minimizing the flowtime should be the scheduler's goal since we aim to reduce the load on the workers to optimize resource utilization and maintain the maximum possible number of available workers. The makespan is the time needed for the system to finish executing the last service [12], which can be calculated as the longest time that any worker from the system takes to finish its assigned services, as given by equation (5). Thus, it is important to minimize this number to ensure completing the average user service in a timely manner. It

is noteworthy that these criteria can be heavily impacted by communities if a generic allocation scheme is used since it can pile workload on a few workers due to their available resources before realizing that those workers are the only available option for unscheduled services.

$$\text{flowtime} = \sum_{i \in A_j} \sum_{j=1}^{m} \frac{e_{i,j}}{n} \tag{4}$$

$$\text{makespan} = \max_{j=1}^{m} \{t_j\} \tag{5}$$

Formulating the problem as a bipartite graph matching renders the Munkres algorithm [15] a suitable solution. The Munkres algorithm, also known as the Hungarian algorithm, is a combinatorial optimization algorithm capable of solving the classical bipartite graph matching which is the assignment problem in polynomial time, specifically with a time complexity of $O(N^3)$. However, in contrast to classical bipartite graph matching, where every vertex from group A is matched with a single vertex from group B, this is not always the optimal case for service allocation. This is since multiple services can be assigned to the same worker within the same cycle. On top of that, the Munkres algorithm is relatively slow, due to its multiple steps and calculations.

To address the aforementioned problems, we need to take the problem a step back to graph matching, better known as the maximum flow algorithm [16]. More specifically, the multi-source multi-sink variation of the problem, where we add an imaginary source that connects to all the sources and an imaginary sink that connect to all sinks. To apply this to bipartite graph matching, we can set the capacity of those new edges to one, limiting the flow to one per source and one per sink (i.e., service). In our case, we want the graph to match the sinks (services) once to avoid assigning redundant work to the workers. However, we strive to allow workers to have multiple services. Thus, we introduce our first parameter. β, which allows the user to set the capacity for the number of services assigned per worker. By default, this can be set to the number of services, so any number of services can be assigned to the same worker if needed.

The maximum flow algorithm with the worker capacity set to one, can maximize the number of matches between workers and services, overriding previous matches if services assigned so far result in a dead-end, where some services are left unmatched. Hence, the maximum flow algorithm can result in the maximum possible number of matching, but they cannot be guaranteed to be the best matches. Replacing the pathfinding portion of the maximum flow algorithm with the shortest path alternative, such as the Bellman-Ford algorithm, elevates it to the minimum cost maximum flow (MCMF) algorithm, which guarantees matches that result in the lowest cost (i.e., service execution time). However, with the cap on source edges at the number of services. The MCMF can assign all services to one worker. The first approach of bipartite graph matching tends to optimize the makespan, while MCMF optimizes the flowtime. CORA bridges the gap between the two approaches. Whenever a service s_i is matched with a worker d_j, we change the cost on the edge from the added "super-source" to d_j from zero to the sum of edge weights for all services assigned to this worker, multiplied by α, as given by equation (6), where α is a tuneable parameter, such that $0 < \alpha < 1$.

$$e_{s,j} = \sum_{i \in A_j} \alpha \cdot e_{i,j} \tag{6}$$

IV. Performance Evaluation

In this section, we evaluate the performance of CORA compared to six prominent heuristic-based resource allocation schemes that could be tweaked to fit into the community-oriented EED-based computing environments. These schemes are Work Queue [13], Min-min [11], Max-min [11], LJFR SJFR [12], and Sufferage [12]. In addition, we compare CORA to the Munkres algorithm [15]. We use the following performance metrics: 1) makespan, 2) flowtime, and 3) runtime.

A. Simulation Setup

The data generated and used in this simulation is the ETC matrix, service set of communities, and worker set of communities. This data is generated using a uniform distribution with some parameters to distinguish between different scenarios. Unless otherwise specified, the number of services and workers is set to 500. In total, we generated and simulated 16 classes of instances by varying the following parameters:

- Worker heterogeneity: represents the variance among the execution times of a given service across all workers. This value can range between 1 and 50 in our simulation. In the average extreme edge computing environment, worker heterogeneity can be considered high due to the wide range of available workers. However, in some cases of corporate-owned devices, they can be relatively close or even identical.
- Service heterogeneity: describes the possible variation among the execution times of services on a given worker, where a low variance indicates that a given worker can run all services with a small execution time gap, while a high indicates a case of vastly dissimilar services. We simulated service heterogeneity between 1 and 50.
- ETC matrix consistency: can have one of two possible values; consistent or inconsistent. In a consistent ETC matrix, a worker d_j executes any service s_i faster than d_k. In this case, worker d_j executes all services faster than worker d_k. In contrast, an inconsistent matrix characterizes the case where worker d_j may be faster than worker d_k for some services and slower for others.
- Community density: defines the number of edges in the graph and the number of unique communities available in the environment at the time of simulation, which translates to the possibility of community overlapping between services and workers. This, in turn, results in matching with fewer constraints. The number of unique

TABLE I
COMPARISON OF STATISTICAL RESULTS ON MAKESPAN AND FLOWTIME FOR 500 SERVICES IN SECONDS.

Instance	Min-min		Max-min		LJFR_SJFR		WorkQueue		Sufferage		CORA		Munkres	
	ms.	ft.	ms.	ft.	ms.	ft.	ms.	ft.	ms.	ft.	ms.	ft.	ms.	ft.
lo-lo-c-s	1014	502	943	513	1016	511	1117	524	2450	**498**	**700**	505	**700**	505
lo-lo-c-d	1034	505	903	512	858	511	973	526	3776	504	**697**	**503**	**697**	**503**
lo-lo-i-s	1137	536	1112	552	1099	550	1232	598	1533	537	**658**	**535**	**658**	**535**
lo-lo-i-d	1116	530	1065	543	1071	539	1219	598	1113	530	**590**	**525**	**590**	**525**
lo-hi-c-s	1053	515	956	528	1012	520	1049	541	3326	**508**	**694**	509	**694**	509
lo-hi-c-d	952	509	948	521	891	518	1075	542	1107	508	**703**	**505**	**703**	**505**
lo-hi-i-s	1231	551	1147	563	1140	565	1272	620	1539	549	**689**	**547**	**689**	**547**
lo-hi-i-d	1120	538	1086	552	1056	547	1202	618	1075	536	**626**	**533**	**626**	**533**
hi-lo-c-s	1072	511	974	525	1012	520	1229	545	1978	**508**	**707**	510	**707**	510
hi-lo-c-d	1040	509	987	524	911	519	1005	541	1399	509	**689**	**506**	**689**	**506**
hi-lo-i-s	1199	546	1143	567	1155	563	1248	615	1618	548	**672**	**545**	**672**	**545**
hi-lo-i-d	1090	537	1072	560	1057	550	1240	617	1126	538	**611**	**532**	**611**	**532**
hi-hi-c-s	2432	**686**	1794	796	1930	746	8479	2062	5137	690	**1735**	714	2230	799
hi-hi-c-d	1793	**622**	1425	713	1786	682	8838	2129	2830	633	**1314**	639	1627	667
hi-hi-i-s	3614	**759**	2089	853	2125	824	9277	2170	2998	773	**2089**	787	**2089**	857
hi-hi-i-d	1709	**683**	1574	762	1629	760	8633	2133	2307	709	**1433**	699	1598	714
Mean	1413	**565**	1201	599	1234	589	3068	961	2207	567	**913**	568	974	581

The instance format is ww-xx-y-z, where ww is service heterogeneity (high/low), xx is worker heterogeneity (high/low), y is matrix consistency (consistent/inconsistent), and z is community density(sparse/dense). The ms. stands for makespan while ft. indicates the flowtime column under each approach

communities, ranges between 10 and 30, while the number of edges per node can vary between 2 and 10.

B. Results and Analysis

Table I shows the makespan and flowtime results obtained from running the scheduler evaluations for every instance. The results shown for every instance are the average of five different simulations. The embedded row clarifies the two columns under each approach. The mean value per approach across all instance classes within the table is shown in the last row. The first column indicates the instance name, in a format that is explained in the note under the table, and the remaining columns indicate the values of makespan followed by flowtime from left to right, respectively for each approach. The highlighted bold value is the minimum value for either makespan or flowtime per instance. As depicted in the table, CORA and Munkres share the minimum makespan across all instances. This is due to their common nature of prioritizing distributing services over different workers and the possibility of reallocating to consider the allocation order that prevents service stacking that significantly increases makespan. However, it is not always desirable to have one service per worker. This idea is highlighted by the instances with high service heterogeneity and service heterogeneity. This can be attributed to the fact that high worker heterogeneity increases the possibility of having workers that are powerful enough to execute multiple services before other workers execute a single one. Another reason is high service heterogeneity, resulting in short services that can be stacked and completed before or close to relatively long services. In those four instances of high worker heterogeneity and high service heterogeneity, we can see CORA's ability to adapt and achieve even lower makespan results. For example, on average, CORA achieves a 24% lead compared to the second-best heuristic. The Munkres algorithm is not considered in this comparison because of its runtime, which will be discussed later. This is in addition to the fact that Munkres lacks the ability to assign multiple services to a worker, significantly reducing its potential for generalization. For the flowtime, most approaches, with the exception of workQueue, are relatively close to each other. However, CORA still outperforms most of the other approaches and comes third within a negligible 0.5% difference behind the best approach on average. Moreover, CORA retains the minimum flowtime for 9 out of 16 instances, which means that in those cases, it is the best option on the two fronts.

Figure 2(a) and Figure 2(b) further illustrate the difference between CORA and the other approaches in average makespan and flowtime, respectively. We can observe the trade-off that other heuristics must make; while the Sufferage and Min-min have a slight lead in average flowtime, they fall behind for makespan. On the other hand, LJFR_SJFR and Max-min retain a relatively low makespan but have to sacrifice some flowtime. Breaking this pattern is the Munkres and CORA. The latter has a considerable lead over all approaches, including the Munkres algorithm. Testing across different data sizes with a varying number of services shows that for all approaches, except for workQueue, the makespan remains consistent, while workQueue experiences a slight increase for larger numbers. Furthermore, the flowtime remains consistent for all approaches with the number of services.

Resource allocation that results in minimum makespan and flowtime is the main goal of the scheduler. However, calculating a near-optimal solution can take a relatively long time, sometimes, to the point where the resulting allocation is no longer relevant as the scheduler runtime exceeds the time the services could have taken in a less optimal solution. Thus, the scheduler runtime is vital to any live allocation approach. WorkQueue sacrifices optimality for speed and simplicity, hence, it has a complexity of $O(NM)$, where N is the number

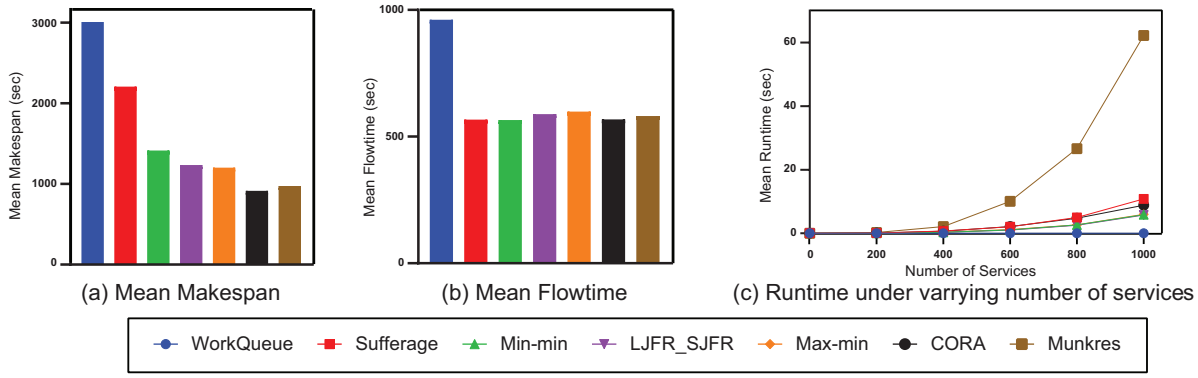

Fig. 2. Performance Results for Min-min, Max-Min, LJFR_SJFR, WorkQueue, Sufferage, CORA, and Munkres.

of services and M is the number of devices. The Min-min, Max-min, LJFR_SJFR, and Sufferage all share a complexity of $O(N^2M)$. CORA has a time complexity of $O((N+M)^2E)$, where E is the number of edges in the formulated graph, which can vary depending on the graph density. Munkres time complexity is $O(6(N+M)^3)$. As depicted in Figure 2(c). The Munkres algorithm's longer runtime is due to factors resulting from the multiple steps that it has to go through before the assignment. In contrast, on average CORA does not encounter the same problem achieving a run time that is up to six times faster than the Munkres algorithm.

V. Conclusion and Future Work

Democratizing the edge by exploiting ample yet underutilized resources of EEDs can open a new tech market for individuals, businesses, enterprises, and municipalities to create their own edge cloud and monetize underused resources. Resource allocation in EED-enabled computing environments is crucial for modern applications, and finding a balance between reduced latency, cost-efficiency, and privacy is challenging. In this paper, we have proposed the Community-Orientated Resource Allocation (CORA) scheme, which utilizes clusters and communities to check all the boxes. CORA recognizes the need for a more complicated approach to retain an acceptable efficiency. CORA has proved to be on par with six other prominent approaches in terms of flowtime and runtime. Additionally, CORA has a 24% better makespan on average in 16 different scenarios of resource allocation with communities. Accommodating potential gains from the cluster scheduler, and the ability to tune parameters, results indicate that CORA is a suitable middle-ground for users prioritizing security and privacy but also seeking high efficiency and utilization of geographically distant devices. In the future, we plan to modify the runtime estimation approach to utilize container history and share knowledge across different services' histories.

Acknowledgment

This research is supported by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC) under grant number ALLRP 549919-20.

References

[1] K. Gyarmathy, "Comprehensive guide to IoT statistics you need to know in 2020," *VXchnge [online]. Tampa, Florida: vXchnge, 2020 (1), 3 [cit. 2020-07-10]. https://www.vxchnge.com/blog/iot-statistics*, 2020.

[2] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[3] M. Gaur and M. Jailia, "Cloud computing data security techniques—a survey," in *Renewable Energy Towards Smart Grid*. Springer, 2022, pp. 55–65.

[4] P. Ranaweera, A. D. Jurcut, and M. Liyanage, "Survey on multi-access edge computing security and privacy," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 1078–1124, 2021.

[5] L. Peterson, T. Anderson, S. Katti, N. McKeown, G. Parulkar, J. Rexford, M. Satyanarayanan, O. Sunay, and A. Vahdat, "Democratizing the network edge," *ACM SIGCOMM Computer Communication Review*, vol. 49, no. 2, pp. 31–36, 2019.

[6] A. Islam, A. Debnath, M. Ghose, and S. Chakraborty, "A survey on task offloading in multi-access edge computing," *Journal of Systems Architecture*, vol. 118, p. 102225, 2021.

[7] "HomeEdge homepage," https://wiki.lfedge.org/display/HOME/Home+Edge+Project, accessed: 2021-10-01.

[8] I. Ahmad, M. G. AlFailakawi, A. AlMutawa, and L. Alsalman, "Container scheduling techniques: A survey and assessment," *Journal of King Saud University-Computer and Information Sciences*, 2021.

[9] R. Zhou, Z. Li, and C. Wu, "Scheduling frameworks for cloud container services," *IEEE/acm transactions on networking*, vol. 26, no. 1, pp. 436–450, 2018.

[10] K. Kaur, S. Garg, G. Kaddoum, S. H. Ahmed, and M. Atiquzzaman, "KEIDS: Kubernetes-based energy and interference driven scheduler for industrial IoT in edge-cloud ecosystem," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4228–4237, 2019.

[11] S. S. Murad and R. Badeel, "Optimized min-min task scheduling algorithm for scientific workflows in a cloud environment," *J. Theor. Appl. Inf. Technol*, vol. 100, pp. 480–506, 2022.

[12] S. Khurana and R. K. Singh, "Survey of scheduling and meta scheduling heuristics in cloud environment," in *Computational Methods and Data Engineering*. Springer, 2021, pp. 363–374.

[13] "Kubernetes homepage," https://kubernetes.io, accessed: 2021-10-01.

[14] J. Dickerson, K. Sankararaman, K. Sarpatwar, A. Srinivasan, K.-L. Wu, and P. Xu, "Online resource allocation with matching constraints," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2019.

[15] W. Fangyang and L. Yuming, "Extended Kuhn-Munkres algorithm for constrained matching search," 2021.

[16] A. Bernstein, M. P. Gutenberg, and T. Saranurak, "Deterministic decremental sssp and approximate min-cost flow in almost-linear time," in *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2022, pp. 1000–1008.