A Scheduling Method for Tasks and Services in IIoT Multi-Cloud Environments

Zhang, Weifan; Kosta, Sokol; Mogensen, Preben

Published in:
Proceedings - 19th International Conference on Distributed Computing in Smart Systems and the Internet of Things, DCOSS-IoT 2023

DOI (link to publication from Publisher):
10.1109/DCOSS-IoT58021.2023.00056

Publication date:
2023

Document Version
Accepted author manuscript, peer reviewed version

A Scheduling Method for Tasks and Services in IIoT Multi-Cloud Environments

1st Weifan Zhang
Aalborg University
Denmark
weifanzh@es.aau.dk

2nd Sokol Kosta
Aalborg University
Denmark
sok@es.aau.dk

3rd Preben Mogensen
Aalborg University
Denmark
pm@es.aau.dk

***Abstract*—Owing to more resources and higher scalability, cloud environments are more prevalent than traditional local servers to deploy Industrial Internet of Things (IIoT) applications. In a multi-cloud environment, there are multiple alternative clouds to deploy applications. These clouds have different resources and network conditions, and at the same time, also the IIoT applications have diverse requirements and priorities. Thus, in order to allocate the necessary resources to applications, there is a need for scheduling algorithms that manage the process of cloud selection and resource allocation. Besides, in practical scenarios, two types of applications can be identified, namely, *services* and *tasks*, with the former being long-lasting executables and the latter being code running for a short amount of time. Even though these two types of applications should be able to be deployed at the same time, to the best of our knowledge, the existing algorithms can only schedule one type of the two. In this paper, we propose an algorithm named Multi-Cloud Application Scheduling Genetic Algorithm (MCASGA), which can schedule both services and tasks at the same time. MCASGA can make scheduling schemes according to application priorities, application dependence, network latency, network bandwidth, CPU, memory, and storage. Our simulated experiments show that in scenarios with different service-to-task ratios, MCASGA outperforms four existing algorithms in the aspects of application acceptance rate, task completion time, and application makespan. The results show that when MCASGA completes 90% of the tasks, other algorithms can only complete less than about 50%.**

***Index Terms*—Industry 4.0, Industrial Internet of Things (IIoT), multi-cloud, service, task, scheduling, Quality of Service (QoS), Genetic Algorithm (GA)**

I. Introduction

The large amount of computation and storage capacity of cloud environments can meet the needs for digitization and automation in Industry 4.0 [1]. Multi-cloud technology enables the integration of heterogeneous clouds with different resources and network conditions [2]. Meanwhile, Industrial Internet of Things (IIoT) applications have different Quality of Service (QoS) requirements and priorities, which means that they are sensitive to different types of resources and their importance levels are also not the same. If an IIoT application is not served by correct resources, it will work abnormally [3]. Therefore, in an IIoT multi-cloud environment, there should be a scheduling algorithm to deploy applications to suitable clouds with the correct types of resources.

This work is fully financed by the project 5G-Robot, Wireless Communication Networks Section, Department of Electronic System, Aalborg University.

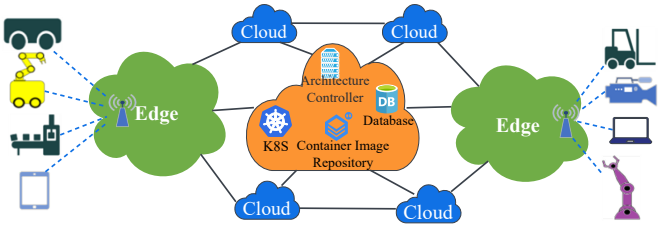

Fig. 1. The multi-cloud architecture running MCASGA.

In practical industrial settings, there are two types of applications: one type runs and occupies resources forever to receive requests and give responses, such as web servers[1] and databases[2], while the other type executes some workloads and releases resources after completion, such as data backups [15]. In this work, we will refer to the former as *services* and to the latter as *tasks*, respectively. In some production scenarios, tasks and services are deployed at the same time. For example, today's most popular cloud application management tool Kubernetes [16] supports running Kubernetes Jobs[3] and Kubernetes Deployments[4] at the same time, which are equivalent to our definition of tasks and services, respectively. Nevertheless, to the best of our knowledge, the cloud application scheduling algorithms proposed by existing research are only aimed at working exclusively either with services or tasks separately, while none of them can schedule services and tasks at the same time. Thus, in this work, we propose MCASGA, an algorithm for managing the scheduling of services and tasks at the same time. MCASGA has the following five characteristics:

(1) **MCASGA supports scheduling both services and tasks at the same time**, as described above.

(2) **MCASGA is based on Genetic Algorithm (GA)**. The cloud application scheduling problem can be formulated as a Multiple Multidimensional Knapsack Problem [17] which is NP-hard, and no algorithm can guarantee finding the optimal solutions of NP-hard problems in polynomial time, so as an approximate algorithm, GA is a popular choice to find the near-optimal solutions.

(3) **MCASGA can not only schedule new applications but also migrate existing ones.** This can optimize resource

[1] https://httpd.apache.org/
[2] https://www.mysql.com/
[3] https://kubernetes.io/docs/concepts/workloads/controllers/job/
[4] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

TABLE I
RELATED WORK SUMMARY

References	Architecture	Management	Workload	Resources	Solution	Optimization
M. Islam et al. 2016 [4]	Cloud	Migration	VM	Cloudlet server	GA	Performance
A. P. Miettinen et al. 2010 [5]	Cloud	Scheduling	Task	Unlimited	Analysis	Energy
L. Yang et al. 2014 [6]	Cloud	Scheduling	Task	VM	Greedy heuristic algorithm	Performance
T. X. Tran et al. 2018 [7]	Multi-cloud	Scheduling	Task	CPU, network latency	Quasi-convex and convex optimization, heuristic algorithm	Performance, energy
T. Soyata et al. 2012 [8]	Multi-cloud	Scheduling	Task	Response time	Greedy algorithm	Performance
Y. Tao et al. 2017 [9]	Cloud	Scheduling, migration	Containerized task	CPU, memory, storage, network bandwidth	Fuzzy inference system	Performance
X. Guan et al. 2016 [10]	Cloud	Scheduling, allocation, scaling	Containerized task	PM, network path length	LP	Cost
M. S. Ajmal et al. 2021 [11]	Cloud	Scheduling	Task	CPU	Hybrid ant genetic algorithm	Performance, cost
C. Guerrero et al. 2018 [12]	Cloud	Scheduling	Containerized service	PM, network distance	GA	Reliability, performance, resource usage
C. Guerrero et al. 2018 [13]	Multi-cloud	Scheduling	Containerized service	VM, network distance	GA	Cost, performance, reliability
F. Legillon et al. 2013 [14]	Multi-cloud	Scheduling	Service	Memory size, memory speed, CPU	GA	Performance
This work	Multi-cloud	Scheduling, migration	Containerized task, Containerized service	CPU, memory, storage, network latency, network bandwidth	GA	Priority-based performance

allocation because the optimal scheduling schemes for the deployed applications may vary when new ones are added to the environment.

(4) MCASGA considers various factors including **application priorities, application dependence, network latency, network bandwidth, CPU, memory, and storage**, which can precisely meet the requirements of applications.

(5) **MCASGA is designed to schedule containerized applications.** Containers with a lower overhead are replacing Virtual Machines (VMs) as the compute instance of cloud application deployments [18].

The main contributions of this paper include:

- We propose a novel algorithm MCASGA to schedule both services and tasks in multi-cloud environments, which is not supported by other existing algorithms.
- We design a fitness function for MCASGA.
- We design two mutation operators for MCASGA.
- We test the performances of 1716 combinations of operators with different parameters in MCASGA, and choose the best combination for MCASGA.
- We conduct simulated experiments to evaluate the performance improvements of MCASGA over existing algorithms.

The rest of this paper is structured as follows. Section II reviews related works. Section III describes MCASGA. Section IV presents the simulated experiments and the results. Section V concludes this paper and discusses future work.

II. RELATED WORK

A roadmap toward Industry 4.0 is defined by [3], in which cloud computing is a major driver of high flexibility and reliability. Paper [2] describes the advantages of multi-cloud, including optimizing costs and QoS, improving the availability of resources and services, and providing backups to deal with disasters. A GA-based model is proposed in [4] to migrate VMs among different cloudlet [19] servers, considering user mobility and the load of servers.

Most existing scheduling methods are designed for *tasks*. The authors of [5] analyze the energy consumption of two mobile devices when executing computation tasks locally or on the cloud. Especially, they use the program execution time and CPU clock speed to calculate the CPU cycles that a task needs to execute, which are used to represent the workload of this task. A greedy heuristic algorithm is proposed in [6] to schedule computation tasks between a cloud with limited resources and multiple user devices, with the objective of minimizing the average completion time of tasks. However, this method only considers cloud resources as servers rather than more specific resources like CPU or memory, and it does not suit multi-cloud scenarios. Another work [7] proposes a joint task offloading and resource allocation approach for multi-cell Mobile-Edge Computing networks. The method is evaluated in the aspects of suboptimality, convergence behavior, and effects of user number, task profile, users' preferences, and inter-cell interference approximation. A mobile-cloudlet-cloud architecture in [8] uses a greedy algorithm to schedule the tasks of face recognition applications based on the response time of cloud servers. Nevertheless, a test in advance is necessary to know the response time of a server for a task, which is not allowed in some scenarios. A fuzzy logic-based container scheduling method is presented in research [9], which splits each workflow into different containerized tasks and schedules the tasks among cloud servers to optimize resource allocation and improve performance. An Linear Programming (LP) method in [10], considering the features of Docker containers, schedules containerized tasks among Physical Machines (PMs) as well as allocates and scales cloud resources. Study [11] divides tasks into groups and adds the feature "pheromone" of Ant Colony Optimization (ACO) into GA to detect loaded VMs, which enables scheduling tasks on less loaded VMs to decrease execution time and data center costs.

There are also works aiming at the scheduling of *services*. In [12], [13], the authors present GA-based algorithms to schedule microservices containers to PMs and VMs respectively; however, both only optimize the new applications, while not able to migrate existing ones. The method in [14] can find a VM provisioning and service placement scheme with the objective of minimizing the cost of VMs considering memory size, memory speed, and CPU power.

Table I shows the comparison of existing research and our approach. In summary, existing works only support scheduling services or tasks. To the best of our knowledge, MCASGA is

the first method to schedule services and tasks together.

III. Methodology

We consider MCASGA in a multi-cloud architecture as presented in Fig. 1. The *architecture controller* is the component that handles the scheduling, which upon receiving requests to deploy different applications runs MCASGA to choose the most suitable cloud for each application and then uses the respective Application Programming Interfaces (APIs) to control the clouds and Kubernetes to manage them. The symbols used in this paper are listed in Table II.

A. Lifecycles of Services and Tasks

MCASGA considers that applications have the lifecycles shown in Fig. 2. If application a is a task, it has six stages, i.e., "wait" (with duration $wt(a)$), "pull image" (with duration $pit(a)$), "input data" (with duration $idt(a)$), "start up" (with duration $sut(a)$), "stable running" (with duration $srt(a)$), and "completed". If a is a service, it does not have the stage "completed", but its "stable running" stage lasts forever. The total duration of the first four stages of a service is its "preparation time", and the total duration of the first five stages of a task is its "completion time", as defined in (1) and (2) respectively. The details of these stages are described in this subsection.

$$pt(a) = wt(a) + pit(a) + idt(a) + sut(a) \tag{1}$$

$$ct(a) = wt(a) + pit(a) + idt(a) + sut(a) + srt(a) \tag{2}$$

wait: Before an application a_i can be deployed, it should wait for four conditions: **a**. all higher-priority services on the same cloud with it enter "stable running" stages, with the duration calculated by (3); **b**. all higher-priority tasks on the same cloud with it enter "completed" stages, with the duration calculated by (4); **c**. all its dependent services enter "stable running" stages, with the duration calculated by (5); **d**. all its dependent tasks enter "completed" stages, with the duration calculated by (6). Thus, the time in the "wait" stage of a_i is calculated by (7).

$$\max_{a\in A, a.c=a_i.c, a^p>a_i^p, a_i.t=False} pt(a) \tag{3}$$

$$\max_{a\in A, a.c=a_i.c, a^p>a_i^p, a_i.t=True} ct(a) \tag{4}$$

$$\max_{d\in a_i^D, d.app.t=False} pt(d.app) \tag{5}$$

$$\max_{d\in a_i^D, d.app.t=True} ct(d.app) \tag{6}$$

$$wt(a_i) = Max[(3),(4),(5),(6)] \tag{7}$$

pull image: After the "wait" stage, a_i should pull its container image from the repository, which means the image should be transmitted from the cloud running the image repository to the cloud running a_i, so the duration is calculated by (8). From this stage, a_i starts occupying resources.

$$pit(a_i) = \frac{a_i^{image}}{dbw(a_i.c, cir)} + rtt(a_i.c, cir) \tag{8}$$

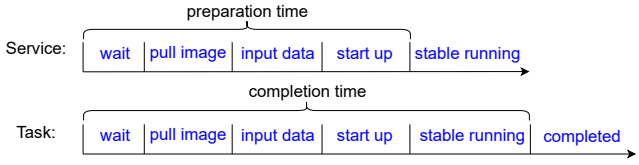

Fig. 2. Lifecycles of services and tasks.

input data: Then, the architecture controller should transmit the input data to a_i, which means the data should be transmitted from the cloud running the architecture controller to the cloud running a_i, so the duration is calculated by (9).

$$idt(a_i) = \frac{a_i^{input}}{dbw(a_i.c, arct)} + rtt(a_i.c, arct) \tag{9}$$

start up: Next, a_i should do some initialization work before stable running. Similar to [5], we use CPU cycles to represent the workloads, so the duration is calculated by (10).

$$sut(a_i) = \frac{a_i^{upcpu}}{a_i.c^{ac} \times a_i.c^{cs}} \tag{10}$$

stable running: After entering this stage, a service will run forever to receive requests and give responses, while a task will execute some workloads and end this stage after completion. If a_i is a task, the time in this stage can be calculated by (11).

$$srt(a_i) = \frac{a_i^{trc}}{a_i.c^{ac} \times a_i.c^{cs}} \tag{11}$$

completed: After a task completes its workloads, it releases the occupied resources and enters this stage. Services do not have this stage and occupy resources forever.

B. Our Proposal: MCASGA

MCASGA is designed based on GA which simulates the evolution of chromosomes and genes of animals and plants. A chromosome consists of many genes, and many chromosomes constitute a population that should experience many generations of evolution. In MCASGA, each chromosome represents a scheduling scheme in which each gene can be either a chosen cloud for an application or "REJ" ("REJ" means this application is rejected and cannot be scheduled to any cloud), so the number of genes in each chromosome should be equal to the number of applications. When using MCASGA to solve a scheduling problem, the user should set CC, IC, SC, MP, and CP firstly. Then MCASGA will generate CC chromosomes as the initial generation of P, and P should experience at most IC generations of evolution to work out the scheduling result. During each generation, the fitness of every $Ch \in P$ is evaluated by a fitness function, and a selection operator, a crossover operator, and a mutation operator are applied on P to generate the next generation. The details of MCASGA are described as follows.

1) Constraints: For a legal scheduling scheme, after it is applied, eqs. (12) to (17) must be met, which guarantees feasible solutions.

When an application is deployed on a cloud, the resources occupied by this application will be reduced from the allocatable resources of the cloud. (12) and (13) mean that

TABLE II
THE SYMBOLS USED IN THIS PAPER.

Symbol	Description
CC	The number of chromosomes in a population
IC	The number of generations of evolution that a population should experience
SC	If the highest fitness value does not change in the latest SC generations, we stop MCASGA and return the best solution so far
MP	The probability of mutation
CP	The probability of crossover
C	The set of clouds
c	A cloud
cir	The cloud running the container image repository
$arct$	The cloud running the architecture controller
c^{ac}	The allocatable CPU logical cores of c
c^{cs}	The clock speed of each CPU logical core of c
c^{am}	The allocatable memory of c
c^{as}	The allocatable storage of c
$ubw(c_i, c_j)$	The allocatable upstream bandwidth between $c_i \in C$ and $c_j \in C$
$dbw(c_i, c_j)$	The allocatable downstream bandwidth between $c_i \in C$ and $c_j \in C$
$rtt(c_i, c_j)$	The RTT between $c_i \in C$ and $c_j \in C$
A	The set of applications to schedule
a	An application
$msp(A)$	The makespan of all $a \in A$
$a.t$	A boolean attribute. $a.t = True$ when a is a task; $a.t = False$ when a is a service
a^{trc}	If $a.t = True$, a^{trc} is the CPU cycles that a needs to execute
$a.new$	A boolean attribute. $a.new = True$ if a is a new application to be scheduled; $a.new = False$ if a was deployed before but needs to be rescheduled now
$a.prevc$	If $a.new = False$, $a.prevc$ records the cloud on which a was running before rescheduling
a^{input}	The input data size of a
a^{image}	The container image size of a
a^{upcpu}	The CPU cycles to be executed during the startup of a
a^{p}	The priority of a
d	If an application depends on another, d is used to describe this dependence relationship
a^D	A set of d, recording all dependent applications of a and the depencence information.
$d.app$	If $d \in a^D$, $d.app$ is an application that a depends on
$d.ubw$	If $d \in a^D$, $d.ubw$ is the upstream bandwidth that the interaction between a and $d.app$ should occupy
$d.dbw$	If $d \in a^D$, $d.dbw$ is the downstream bandwidth that the interaction between a and $d.app$ should occupy
$d.rtt$	If $d \in a^D$, $d.rtt$ is the highest RTT that can support the interaction between a and $d.app$
$a.c$	$a.c$ is the cloud on which a is scheduled. $a.c = REJ$ means a is rejected
$wt(a)$	The time in "wait" stage of a
$pit(a)$	The time in "pull image" stage of a
$idt(a)$	The time in "input data" stage of a
$sut(a)$	The time in "start up" stage of a
$srt(a)$	If $a.t = True$, $srt(a)$ is the time in "stable running" stage of a
$pt(a)$	If $a.t = False$, $pt(a)$ is preparation time of a
$ct(a)$	If $a.t = True$, $ct(a)$ is completion time of a
P	A population which is a set of chromosomes
Ch	A chromosome which is a set of genes
t^{ceil}	The ceiling execution time used in the fitness function

applications cannot occupy resources more than the capacity of any clouds.

$$\forall c \in C,\ c^{ac} \geq 0 \ \textbf{and}\ c^{am} \geq 0 \ \textbf{and}\ c^{as} \geq 0 \tag{12}$$

$$\forall c_i, c_j \in C,\ ubw(c_i, c_j) \geq 0 \ \textbf{and}\ dbw(c_i, c_j) \geq 0 \tag{13}$$

(14) means that if two applications with a dependence relationship are scheduled on two clouds, the RTT between the two clouds should be lower than the required RTT of the two applications.

$$\forall a \in A, \forall d \in a^D,\ rtt(a.c, d.app.c) \leq d.rtt \tag{14}$$

An application cannot run well without its dependent applications, so if any of its dependent applications cannot be deployed, it should also be rejected, as constrained in (15).

$$\forall a \in A, \forall d \in a^D,\ \textbf{if}\ d.app.c = REJ\ \textbf{then}\ a.c = REJ \tag{15}$$

While scheduling new applications, MCASGA also reschedules the deployed applications to optimize resource allocation, but MCASGA does not reject any deployed applications, as constrained in (16). Moreover, during the rescheduling, only services are allowed to be migrated to different clouds, because a task cannot continue its work without the working states stored on the previous cloud, so we have the constraint (17).

$$\forall a \in A,\ \textbf{if}\ a.new = False\ \textbf{then}\ a.c \neq REJ \tag{16}$$

$$\forall a \in A,\ \textbf{if}\ a.new = False\ \ \textbf{and}\ \ a.t = True\ \ \textbf{then}\ a.c = a.prevc \tag{17}$$

2) Fitness Function: Fitness is used to evaluate the quality of solutions. In practical production, users hope that services enter the "stable running" stage as soon as possible and tasks enter the "completed" stage as soon as possible. Therefore, the objective of MCASGA is to minimize the preparation times of services (calculated by (1)) and the "completion times of tasks (calculated by (2)). For $\forall a \in A$, the higher $pt(a)$ or $ct(a)$, the lower a contributes to the fitness. Based on this objective, we design the fitness function shown in Algorithm 1, in which the higher the fitness value, the better the chromosome. This fitness function iterates over all applications and calculates their contributions to fitness. Applications with higher priorities have higher weights (bigger impacts), and the rejected applications do not contribute to fitness. Algorithm 1 needs an input parameter t^{ceil}, which represents the ceiling execution time of applications. The ideal t^{ceil} should be larger than the makespan of all applications calculated by (18) but not too large. MCASGA uses Algorithm 2 to determine t^{ceil} after the initialization step described in III-B3. In Algorithm 2, the makespan of the applications deployed on a cloud c is calculated by (19).

$$msp(A) = Max\left(\max_{a \in A,\, a.t=False} pt(a), \max_{a \in A,\, a.t=True} ct(a)\right) \tag{18}$$

$$msp(c) = Max\left(\max_{a \in A, a.c=c,\, a.t=False} pt(a), \max_{a \in A, a.c=c,\, a.t=True} ct(a)\right) \tag{19}$$

3) Initialization: At the beginning of MCASGA, we randomly generate CC chromosomes that meet the constraints of eqs. (12) to (17) as the initial population.

Algorithm 1 Fitness Function

Input: C, A, Ch, t^{ceil};
Output: Ch's fitness value $f(Ch)$;

$f(Ch) \leftarrow 0$, $f^{this} \leftarrow 0$
for each $a \in A$ **do**
  if $a.c = REJ$ **then**
    continue
  end if
  if $a.t = True$ **then**
    $f^{this} \leftarrow (t^{ceil} - ct(a)) \times a^p$
  else
    $f^{this} \leftarrow (t^{ceil} - pt(a)) \times a^p$
  end if
  if $f^{this} < 0$ **then**
    $f^{this} \leftarrow 0$
  end if
  $f(Ch) \leftarrow f(Ch) + f^{this}$
end for
return

Algorithm 2 Ceiling Execution Time

Input: C, A, initial population P^{init};
Output: t^{ceil};

$mspcs \leftarrow \{new\ empty\ array\}$
for each $Ch \in P^{init}$ **do**
  for each $c \in C$ **do**
    $mspcs$.append($msp(c)$)
  end for
end for
Sort($mspcs$)
Delete largest 10% and smallest 10% elements in $mspcs$
$amsp \leftarrow$ average value in $mspcs$
$mmsp \leftarrow$ max value in $mspcs$
$t^{ceil} \leftarrow 2 \times Min(mmsp \times 1.5, amsp \times 4)$
return

4) Selection Operator: Selection operators select the chromosomes with better fitness from one generation to create the next generation. We test roulette wheel selection [11], [20] and binary tournament selection [13] in Subsection IV-C, choosing binary tournament selection for MCASGA. In binary tournament selection, every time we randomly pick two chromosomes from the old generation and add the one with the higher fitness value calculated by Algorithm 1 to the new generation.

5) Crossover Operator: Crossover operators are used to create offspring chromosomes from parent chromosomes. As described in Subsection IV-C, we test one-point crossover and two-point crossover [21], choosing the former for MCASGA. Fig. 3 shows the details of one-point crossover. Every two chromosomes have a CP probability to do a crossover.

6) Mutation Operator: Mutation can introduce new features to the population to reduce the possibility of falling into local optima. We design two mutation operators:

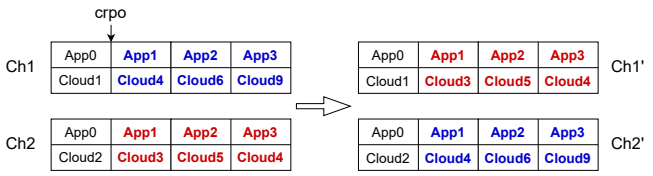

Fig. 3. For two chromosomes, one-point crossover operator randomly chooses a crossover point $crpo$ and interchange the genes after $crpo$.

Algorithm 3 MCASGA

Input: C, A;
Output: Scheduling result Ch_{res};

Initialize P (III-B3)
Determine t^{ceil} by Algorithm 2
Run Algorithm 1 to calculate the fitness value of every $Ch \in P$
Apply selection operator (III-B4) on P
for $generation \leftarrow 1$ *to* IC **do**
  Apply crossover operator (III-B5) on P
  Apply mutation operator (III-B6) on P
  Run Algorithm 1 to calculate the fitness value of every $Ch \in P$
  Apply selection operator (III-B4) on P
  if the highest fitness value does not change in the latest SC generations **then**
    break
  end if
end for
$Ch_{res} \leftarrow$ the chromosome with the highest fitness value
return

- **Chromosome-based mutation**: Every chromosome has an MP probability to be replaced with a randomly generated new chromosome that can meet eqs. (12) to (17).
- **Gene-based mutation**: Every gene has an MP probability to be replaced with a randomly generated new gene. After the replacement, if the chromosome of this gene cannot meet eqs. (12) to (17), we replace it with a randomly generated new chromosome that can meet eqs. (12) to (17).

In Subsection IV-C, we test the performance of the two mutation operators and choose gene-based mutation in MCASGA.

Algorithm 3 is the whole process of MCASGA.

IV. Evaluation

We conduct our experiments by simulation. The algorithm implementation code, the experiments code, and the data are available at this repository[5].

[5]https://bitbucket.org/weifanzhang/gogeneticwrsp/src/main/

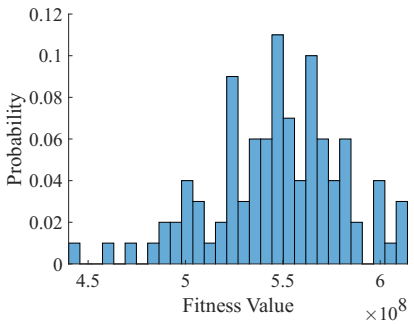

Fig. 4. Probability distribution of fitness values of MCASGA results.

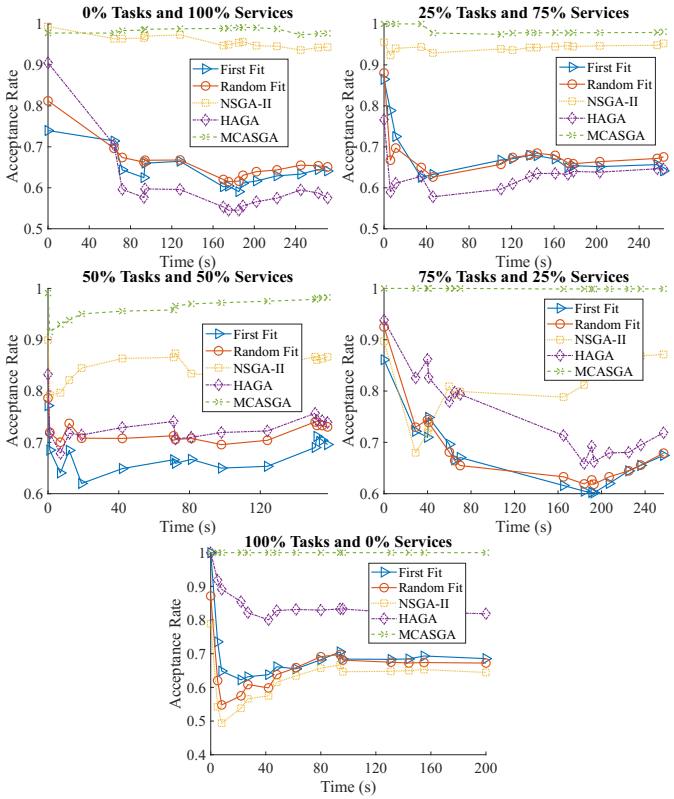

Fig. 5. Priority weighted application acceptance rate.

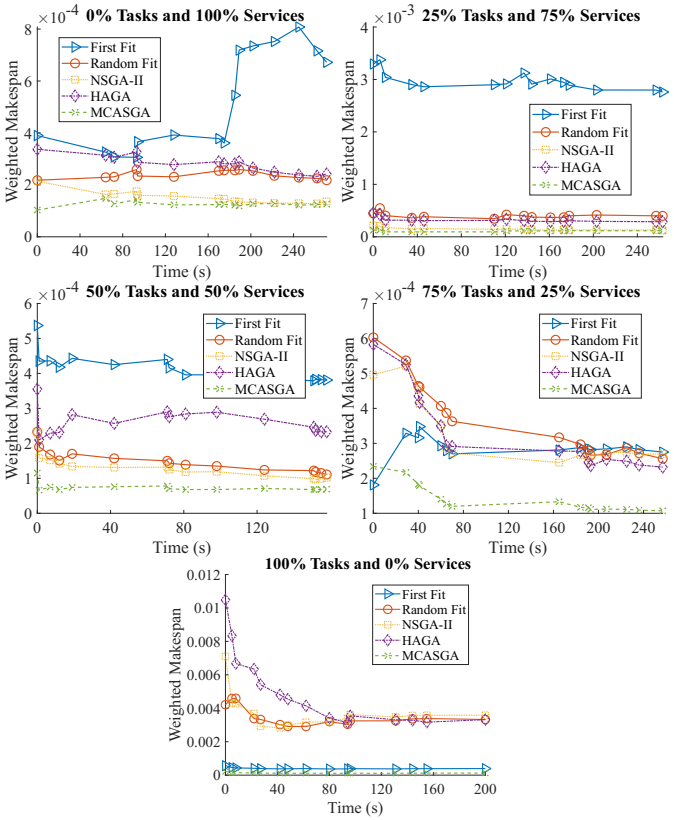

Fig. 6. Priority weighted makespan.

A. Simulation Parameters

In our experiments, cloud resources are generated as follows:

- **CPU**: Randomly chosen from the parameters of AMD EPYC 7002 and 7003 series.
- **Memory**: Randomly chosen from the parameters of research [4] and servers in the online shop Servermall.
- **Storage**: Randomly chosen from the parameters of research [4] and Seagate BarraCuda 2.5" Hard Drive.
- **Network**: We measure the networks among an Aalborg University campus and other servers[6], combining them with the parameters used by [8] to generate our parameters. The bandwidth values (unit: Mb/s) are generated by "$X \sim N(145.23, 215.04^2)$, $s.t.$ $0.873 \leq X \leq 935$". The RTT values (unit: ms) are generated by "$X \sim N(30, 10.44^2)$, $s.t.$ $20 \leq X \leq 40$", "$X \sim N(4488.5, 6654.29^2)$, $s.t.$ $110 \leq X \leq 18000$", "$X \sim N(2.55, 1.51^2)$, $s.t.$ $0.508 \leq X \leq 5.571$", and "$X \sim N(181.29, 90.76^2)$, $s.t.$ $45.19 \leq X \leq 324.43$".

Application requirements are generated as follows:

- **Service CPU**: CPU frequency values (unit: GHz) that services need to occupy are generated by "$X \sim N(3.91, 3.46^2)$, $s.t.$ $1.00 \leq X \leq 14.80$" based on research [9], [14], and AMD EPYC 7002 and 7003 series.
- **Task CPU**: CPU cycles that tasks need to execute are generated by "$X \sim U(157482134186.67, 289910292480.00)$" [9].
- **Memory**: Randomly chosen in $\{1, 1, 2, 1, 0.49\}$ (unit: GB) [9], [14].
- **Storage**: Randomly chosen in $\{8, 4, 3, 2\}$ (unit: GB) [9].
- **Input data**: Generated by "$X \sim U(430080, 65011712)$" (unit: B) [8], [9].
- **Image size**: Generated by "$X \sim N(343125649.8, 322164492.5^2), s.t. 13619.2 \leq X \leq 1043333120$" (unit: B) according to Docker Hub images.
- **Startup CPU cycles**: Generated by "$X \sim N(3732231137, 880522502.9^2)$, $s.t.$ $2888508672 \leq X \leq 4645436627$" [22].
- **Priority**: Generated by "$X \sim U(1, 65535)$".
- **Dependence**: Each application has the possibility of $\frac{4}{7}$ to depend on a number (randomly chosen in $\{1, 2, 5, 3, 1, 1, 1, 2\}$) of higher-priority applications [12].
- **Network**: The bandwidth values (unit: Mb/s) are randomly chosen in $\{0.0098, 0.098, 1.0, 20.0, 1.0, 204.8, 2048.0, 10.0, 100.0\}$, and the RTT values (unit: ms) are randomly chosen in $\{2000, 2000, 2, 1000, 10, 20, 10, 10, 1\}$ [3].

B. Weaken the Influence of Random Factors

For one scheduling problem, MCASGA may give different solutions because of the random factors in GA. In order to weaken the influence of random factors in our evaluation, we repeat every experiment ten times and calculate the average value of each metric similar to [11]. In this subsection, we validate the effectiveness of average values.

We run MCASGA 100 times to schedule 40 applications to 5 clouds. The fitness values of the 100 results have the probability distribution shown in Fig. 4, which is similar to a

[6] https://github.com/R0GGER/public-iperf3-servers/

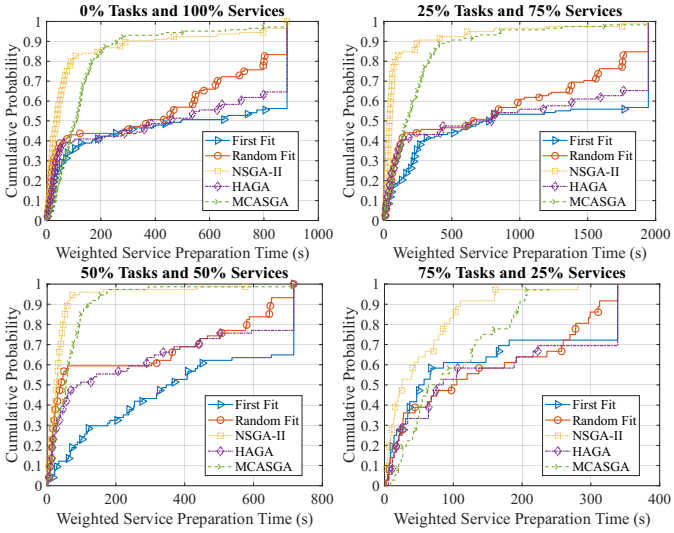

Fig. 7. Priority weighted service preparation time.

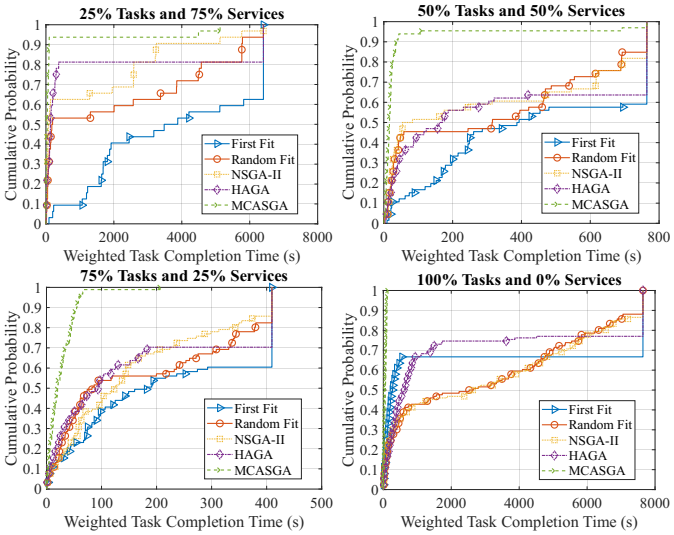

Fig. 8. Priority weighted task completion time.

normal distribution, so average values can effectively weaken random factor influence.

C. MCASGA Operators

Different GA operators bring different performances. In order to optimize MCASGA, we test all combinations of "two-point crossover with CP in {0.0, 0.1, ..., 1.0} **vs.** one-point crossover with CP in {0.0, 0.1, ..., 1.0}", "binary tournament selection **vs.** roulette wheel selection", "chromosome-based mutation with MP in about {0.0, 0.05, ..., 1.0} **vs.** gene-based mutation with MP in about {0.001, 0.002, ..., 0.02}". There are 1716 combinations, and with every combination, we run MCASGA to schedule 40 applications to 5 clouds. In the 1716 results, the combination "binary tournament selection, one-point crossover with $CP = 0.4$, and gene-based mutation with $MP = 0.003$" has the highest fitness value, as shown in Table III, so we use these operators for MCASGA in our following experiments.

D. Experiments and Metrics

This subsection demonstrates our simulated experiments. We generated 10 clouds and 15 groups of applications. Every application is only possible to depend on the applications in the same group with it. The 15 groups need to be deployed at different times, and the time intervals (unit: s) between each group follow the exponential distribution "$X \sim E(\frac{1}{15}),\ s.t.\ X \geq 1$", which means the experiment is a Poisson process and the average time interval between two groups is 15s. Our experiments have five scenarios with different proportions (from 0% to 100%) of tasks and services. We set $CC = 200$, $IC = 5000$, and $SC = 250$ for MCASGA.

MCASGA is compared with four algorithms, i.e., First Fit (choosing the first solution), Random Fit (randomly choosing a solution), NSGA-II [13], and HAGA [11], considering metrics:

- **Priority Weighted Application Acceptance Rate**: When resources are not sufficient, a good algorithm should reject low-priority applications and serve high-priority ones better, so we have this metric calculated by (20).
- **Priority Weighted Service Preparation Time**: Calculated by (21), this metric can reflect services performance.
- **Priority Weighted Task Completion Time**: Calculated by (22), this metric can reflect tasks performance.
- **Priority Weighted Makespan**: Calculated by (23), it shows the comprehensive performance of services and tasks.

$$\frac{\sum_{a \in A, a.c \neq REJ} a^p}{\sum_{a \in A} a^p} \tag{20}$$

$$pt(a) \times a^p \times 0.00001,\ s.t.\ a \in A, a.t = False \tag{21}$$

$$ct(a) \times a^p \times 0.00001,\ s.t.\ a \in A, a.t = True \tag{22}$$

$$\frac{msp(A)}{\sum_{a \in A, a.c \neq REJ} a^p} \tag{23}$$

E. Results

Fig. 5 compares the priority weighted application acceptance rate, in which MCASGA has a significant advantage over other algorithms, keeping the percentage higher than 90% in all scenarios, which means that MCASGA can use the same resources to run more high-priority applications. This is because high-priority applications can contribute higher fitness in MCASGA, while other algorithms do not consider application priority. Another reason is that only MCASGA can migrate or pause deployed applications to optimize resource allocation. In the other four algorithms, NSGA-II performs better when services have a big proportion, and HAGA performs better in the 100%-task scenario, as they are designed to optimize services and tasks respectively.

With regard to the execution times of applications, MCASGA has the shortest weighted makespan under all service-to-task ratios as presented in Fig. 6, because it aims to shorten the execution time of both services and tasks, while other algorithms only consider one type. Fig. 7 illustrates that MCASGA has less weighted service preparation time than three algorithms, but NSGA-II performs better than it. The reason is that NSGA-II only optimizes the scheduling of services, but MCASGA makes trade-offs between services and tasks, resulting in the sacrifice of services. However, the trade-offs are worthwhile because of the higher acceptance

TABLE III
BIGGEST 10 FITNESS VALUES OF 1716 DIFFERENT OPERATOR COMBINATIONS.

Ranking	Selection operator	Crossover operator	Crossover probability	Mutation operator	Mutation probability	Fitness value
1	binary tournament	one-point	0.4	gene-based	0.003	5.840449909361554e+08
2	binary tournament	two-point	1	gene-based	0.017	5.805549164372113e+08
3	binary tournament	two-point	0.8	gene-based	0.016	5.790506593264892e+08
4	binary tournament	one-point	0.5	gene-based	0.018	5.768529743158308e+08
5	binary tournament	one-point	1	gene-based	0.008	5.766490686347214e+08
6	binary tournament	two-point	0.9	gene-based	0.009	5.7657747804722e+08
7	binary tournament	two-point	1	gene-based	0.008	5.744743876787698e+08
8	binary tournament	one-point	0.2	gene-based	0.012	5.733341633030124e+08
9	binary tournament	one-point	0.8	gene-based	0.01	5.726034482470758e+08
10	binary tournament	two-point	1	gene-based	0.013	5.723139889526731e+08

rate in Fig. 5, the lower weighted makespan in Fig. 6, and the evidently lower weighted task completion time in Fig. 8. Fig. 8 shows that when MCASGA completes 90% of the tasks, other algorithms can only complete less than about 50%. Although HAGA focuses on optimizing tasks, it divides tasks into groups and schedules each group separately, which does not show satisfying effects without a global view.

V. CONCLUSION

This paper proposes a novel algorithm named MCASGA to schedule both services and tasks in multi-cloud environments to improve the performance of IIoT applications by allocating more resources to high-priority applications, taking into account application priorities, application dependence, network latency, network bandwidth, CPU, memory, and storage. To the best of our knowledge, MCASGA is the first algorithm that can schedule services and tasks at the same time. The results of the simulated experiments indicate that MCASGA performs better than First Fit, Random Fit, NSGA-II, and HAGA in the aspects of application acceptance rate, task completion time, and application makespan. The results show that when MCASGA completes 90% of the tasks, other algorithms can only complete less than about 50%.

In future research, firstly we plan to conduct our experiments in actual production scenarios. Secondly, the calculation time of MCASGA to get scheduling schemes needs to be shortened in the next stage. Thirdly, we will add a new feature to MCASGA to schedule IIoT applications not only to clouds but also to VMs inside clouds.

REFERENCES

[1] B. Chen, J. Wan, L. Shu, P. Li, M. Mukherjee, and B. Yin, "Smart factory of industry 4.0: Key technologies, application case, and challenges," *Ieee Access*, vol. 6, pp. 6505–6519, 2017.

[2] J. Hong, T. Dreibholz, J. A. Schenkel, and J. A. Hu, "An overview of multi-cloud computing," in *Workshops of the international conference on advanced information networking and applications*, pp. 1055–1068, Springer, 2019.

[3] I. Rodriguez, R. S. Mogensen, A. Schjørring, M. Razzaghpour, R. Maldonado, G. Berardinelli, R. Adeogun, P. H. Christensen, P. Mogensen, O. Madsen, *et al.*, "5g swarm production: Advanced industrial manufacturing concepts enabled by wireless automation," *IEEE Communications Magazine*, vol. 59, no. 1, pp. 48–54, 2021.

[4] M. Islam, A. Razzaque, and J. Islam, "A genetic algorithm for virtual machine migration in heterogeneous mobile cloud computing," in *2016 International Conference on Networking Systems and Security (NSysS)*, pp. 1–6, IEEE, 2016.

[5] A. P. Miettinen and J. K. Nurminen, "Energy efficiency of mobile clients in cloud computing," in *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[6] L. Yang, J. Cao, H. Cheng, and Y. Ji, "Multi-user computation partitioning for latency sensitive mobile cloud applications," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2253–2266, 2014.

[7] T. X. Tran and D. Pompili, "Joint task offloading and resource allocation for multi-server mobile-edge computing networks," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 1, pp. 856–868, 2018.

[8] T. Soyata, R. Muraleedharan, C. Funai, M. Kwon, and W. Heinzelman, "Cloud-vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture," in *2012 IEEE symposium on computers and communications (ISCC)*, pp. 000059–000066, IEEE, 2012.

[9] Y. Tao, X. Wang, X. Xu, and Y. Chen, "Dynamic resource allocation algorithm for container-based service computing," in *2017 IEEE 13th International Symposium on Autonomous Decentralized System (ISADS)*, pp. 61–67, IEEE, 2017.

[10] X. Guan, X. Wan, B.-Y. Choi, S. Song, and J. Zhu, "Application oriented dynamic resource allocation for data centers using docker containers," *IEEE Communications Letters*, vol. 21, no. 3, pp. 504–507, 2016.

[11] M. S. Ajmal, Z. Iqbal, F. Z. Khan, M. Ahmad, I. Ahmad, and B. B. Gupta, "Hybrid ant genetic algorithm for efficient task scheduling in cloud data centers," *Computers and Electrical Engineering*, vol. 95, p. 107419, 2021.

[12] C. Guerrero, I. Lera, and C. Juiz, "Genetic algorithm for multi-objective optimization of container allocation in cloud architecture," *Journal of Grid Computing*, vol. 16, no. 1, pp. 113–135, 2018.

[13] C. Guerrero, I. Lera, and C. Juiz, "Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications," *The Journal of Supercomputing*, vol. 74, no. 7, pp. 2956–2983, 2018.

[14] F. Legillon, N. Melab, D. Renard, and E.-G. Talbi, "Cost minimization of service deployment in a multi-cloud environment," in *2013 IEEE Congress on Evolutionary Computation*, pp. 2580–2587, IEEE, 2013.

[15] K. Bohora, A. Bothe, D. Sheth, R. Chopade, and V. Pachghare, "Backup and recovery mechanisms of cassandra database: A review," *Journal of Digital Forensics, Security and Law*, vol. 15, no. 2, p. 5, 2021.

[16] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE cloud computing*, vol. 1, no. 3, pp. 81–84, 2014.

[17] Y. Song, C. Zhang, and Y. Fang, "Multiple multidimensional knapsack problem and its applications in cognitive radio networks," in *MILCOM 2008-2008 IEEE Military Communications Conference*, pp. 1–7, IEEE, 2008.

[18] V. Medel, O. Rana, J. Á. Bañares, and U. Arronategui, "Adaptive application scheduling under interference in kubernetes," in *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, pp. 426–427, IEEE, 2016.

[19] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[20] S. Deng, L. Huang, J. Taheri, and A. Y. Zomaya, "Computation offloading for service workflow in mobile cloud computing," *IEEE transactions on parallel and distributed systems*, vol. 26, no. 12, pp. 3317–3329, 2014.

[21] T. Gwiazda, "Genetic algorithms reference vol. i. crossover for single-objective numerical optimization problems [online]. tomaszgwiazda e-books," 2006.

[22] B. Xavier, T. Ferreto, and L. Jersak, "Time provisioning evaluation of kvm, docker and unikernels in a cloud platform," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 277–280, IEEE, 2016.