GA-Par: Dependable Microservice Orchestration Framework for Geo-Distributed Clouds

Zhenyu Wen, Tao Lin, Renyu Yang, *Member, IEEE*, Shouling Ji, *Member, IEEE*, Rajiv Ranjan, *Senior Member, IEEE*, Alexander Romanovsky, Changting Lin, Jie Xu, *Member, IEEE*

Abstract—Recent advances in composing Cloud applications have been driven by deployments of inter-networking heterogeneous microservices across multiple Cloud datacenters. System dependability has been of the upmost importance and criticality to both service vendors and customers. Security, a measurable attribute, is increasingly regarded as the representative example of dependability. Literally, with the increment of microservice types and dynamicity, applications are exposed to aggravated internal security threats and externally environmental uncertainties. Existing work mainly focuses on the QoS-aware composition of native VM-based Cloud application components, while ignoring uncertainties and security risks among interactive and interdependent container-based microservices. Still, orchestrating a set of microservices across datacenters under those constraints remains computationally intractable. This paper describes a new dependable microservice orchestration framework GA-Par to effectively select and deploy microservices whilst reducing the discrepancy between user security requirements and actual service provision. We adopt a hybrid (both whitebox and blackbox based) approach to measure the satisfaction of security requirement and the environmental impact of network QoS on system dependability. Due to the exponential grow of solution space, we develop a parallel Genetic Algorithm framework based on Spark to accelerate the operations for calculating the optimal or near-optimal solution. Large-scale real world datasets are utilized to validate models and orchestration approach. Experiments show that our solution outperforms the greedy-based security aware method with 42.34% improvement. GA-Par is roughly 4x faster than a Hadoop-based genetic algorithm solver and the effectiveness can be constantly guaranteed under different application scales.

Index Terms—service orchestration, dependability, microservice

1 Introduction

Microservices are excellent building blocks for composing applications whose components are distributed across multiple Cloud datacenters [1] for agility, performance and fault-tolerance [2] [3] [4]. The strengths of microservices help enable distributed application deployments by allowing quick provisioning and updating across geo-distributed Clouds. For example, a recent study based on benchmark Docker Containers on AWS EC2 showed that they need much less time to start or boot up (less than 50 milliseconds) as compared to Virtual machines (VMs) (30-45 seconds) as well as having no or negligible memory overhead [5]. Moreover, the containerized microservices provide considerable freedom in personalizing users' preferences. For example, Docker Hub has stored over ten million images that are available for users to create their customized applications. Therefore, the diversity significantly increases the configuration dimension of each single microservice as well as the scale of the search space to find a suitable microservice compared to the VM-based Cloud services.

Dependability is a key system concern that incorporates attributes of reliability, availability, safety, integrity, maintainability, etc. Meanwhile security is increasingly recognized as an important factor of dependability and brings in considerations of confidentiality, availability and integrity [6]. Given the abstract nature of dependability, we can take security as an example and measurable attribute. Microservice orchestration is the procedure of determining and selecting best microservice instances to compose an application workflow that satisfy application's functional and nonfunctional requirements QoS. In this context, system dependability will be influenced by both internal and external causes. Internally, topological layout and the security attributes' satisfaction quantitatively determine the consequent dependability. Externally, unpredictable environmental variables such as network jitter and noisy neighbors [7] also impact the dependability across geo-distributed Clouds. This is due to the logical satisfaction or optimization that is achieved by internal examination may no longer exist. For instance, [8] [9] report that network QoS uncertainties of Cloud-hosted services have been widely recognized and it is extremely difficult to accurately capture such uncertainties. Therefore, to achieve a holistically dependable orchestration necessitates both internally quantitative measurement and environmental considerations.

Traditionally, security experts decide the service component placement based on their domain knowledge to guarantee the overall application security level [10]. However, such method heavily relies on human expertise and fails to deploy in large-scale applications if the application consists of a huge number of service components. Thus, a generic mechanism to express the security risks and quantitatively measure the security level of a microservice is urgently

- *Z.Wen, R. Ranjan and A.Romanovsky are with Newcastle University, United Kingdom. E-mail: {zhenyu.wen, Raj.Ranjan, alexander.romanovsky}@newcastle.ac.uk*
- *T.Lin is with the EPFL, Switzerland. E-mail: tao.lin@epfl.ch*
- *R.Yang and J.Xu are with University of Leeds, UK and BDBC, Beihang University, China. E-mail: renyu.yang@edgetic.com (corresponding author), j.xu@leeds.ac.uk.*
- *S. Ji and C.Lin are with Zhejiang University, China. E-mail:{sji, linchangting}@zju.edu.cn*

Manuscript received ???; revised ???

required. However, existing security-aware algorithms for service composition randomly compute or statically determine the security level by simply checking the adopted means such as the encryption algorithm or security policy, etc. [11][12]. This will result in decreased accuracy in the context of massive-scale microservices. Also existing works merely regard the dependability as mathematical optimization ignore the exploiting the external system uncertainties, resulting in less realistic orchestration. Finally, the explosively increasing number of workflow scale and microservice candidates that available for application composition leads to computation efficiency issues by using traditional approaches [13] [14].

In this paper, we propose GA-Par framework to provision a dependable microservice orchestration and deal with the involved computation efficiency issue. It aims to provide an optimal solution of deploying microservice workflow across geo-distributed datacenters. We take the orchestration as an optimization problem to maximize the dependability satisfaction in terms of security requirement and network QoS. We adopt a hybrid (both whitebox and blackbox based) approach to measure relevant factors. Internally, we propose a topology-aware security satisfaction model from whitebox perspective to minimize the discrepancy between the user's requirements and actual service provision. We exploit the discrepancy of individual microservices and adjunct communications between microservices to reach logical optimization. To achieve this, we comprehensively analyze the security technologies, pertaining risks and the target security goals to generate a quantitative measure of the security satisfactory levels. Regarding environmental uncertainties that may influence dependability, we develop a statistical model considering transmission rates and service response time in a black-box manner. This will be added upon the security satisfaction model to formulate a holistic optimization problem for an actually improved dependability. Finally, we develop a novel parallel genetic algorithm framework, GA-Par based on Apache Spark, to effectively find the sub-optimal solution. A multi-phase hybrid parallelization management can adaptively enable both fitness evaluation and population reproduction within the genetic algorithm to reach the maximized parallelism. Experiments show that GA-Par can fulfill up to 33x times acceleration of the execution time compared against other parallelization approaches and the effectiveness can be constantly guaranteed under different application scales. The main contributions of this work are:

- A mathematic optimization model to formulate the dependable microservice orchestration considering both internal security requirement satisfaction and environmental uncertainty factors.
- A measurement to quantify the satisfactory level of security goals by exploiting security risks and techniques among interdependent microservices.
- A data-driven statistical model describe the network QoS of Cloud based containerized microservices.
- A multi-phase parallelization framework to accelerated the execution of Genetic Algorithms, that can significantly improve orchestration efficiency.

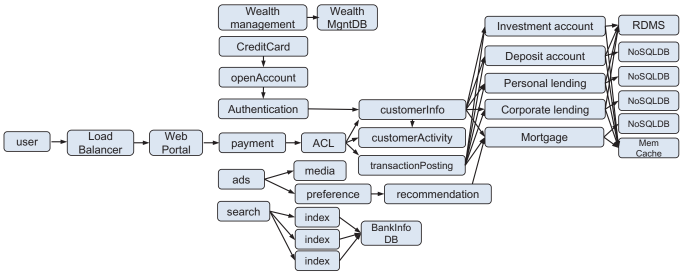


Fig. 1. e-banking microservices (a representative case [15])

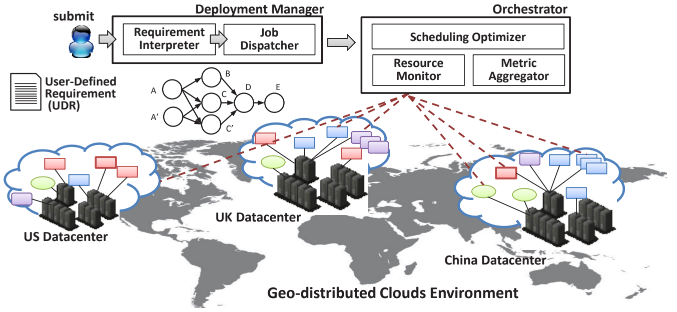


Fig. 2. Microservice orchestration in a geo-distributed Cloud environment.

Organization. We firstly present the motivation and requirements in Section 2. We describe the topology-aware security satisfaction measurement in Section 3 and the external uncertainty model in Section 4. In section 5, we formalize the problem as an optimization problem and then develop a parallel framework to solve it in Sections 6. Experiments are shown in Section 7. Following a review of related work in Section 8, the paper draws conclusions and outlines further works.

2 Motivation

Geo-distributed Microservices. Fig 1 describes an e-banking example, which is one of representative applications that adopt microservice architecture [15]. Regarding the front-end, a logined user can process a payment from their account, pay or request the credit cards, browse information and request loans or mortgage, and acquire wealth management services. The back-end data-related microservices consist of in-memory or persistent databases and relational database that underpin the information system and services. In the financial and banking scenario, dependability is a primary consideration not only in security guarantee in a single microservice (e.g., the privacy protection in lending and mortgage, the confidentiality in in credit card, the availability in database access, etc.), but the secure data transmission between microservices. For further resiliency and high quality of service, a global service provider can spread the microservices over different data centers or clouds. By leveraging geographically distributed architecture (shown in Fig. 2), if one has serious connectivity or late-timing issues, the holistic service can still operate in other replicas. In this paper, we propose a microservices orchestration framework which ensures the dependability of composed microservices across geo-distributed Clouds.

Dependability and security. The original definition of dependability is defined as accepted dependence and the ability to deliver service that can justifiably be trusted. Traditional dependability for a system incorporates availability, reliability, maintainability. Nowadays, applying security measures to the appliances of a system generally improves the dependability by limiting the number of externally originated errors [6]. Therefore, reducing or mitigating security risks can reduce the consequences of unforeseen dependability threats. The measurement of security level plays a significantly important role in quantifying how well a service functions dependably in the presence of faults or attacks over a specified period of time.

Characteristics in geo-distributed and containerized environment. A geo-distributed application consists of a set of microservices with different functionalities. Each type of microservice has a huge number of candidates with diverse configurations in terms of resource request, network configuration, QoS requirement, etc. Due to the deviations of sensitivity to network environment among different microservices, different scheme of microservice placement in geo-distributed environments also affects the overall dependability. We summarize the following characteristics that motivate this work:

- **[C1]** *Customers are exposed to a variety of security risks.* For example, main sources of security threats include human-caused sabotage of network, malicious attacks, etc. Hence, the achievability of security goals is highly affected by how microservices can mitigate or avoid potential security risks.
- **[C2]** *Differentiated security-levels are provisioned by different microservices.* There are more than ten million images stored in Docker Hub. A lot of images share the same functionalities, but with different non-functional configurations such as security. For example, the seven most well known cryptographic algorithms are: DES, RSA, HASH, MD5, AES, SHA-1 and HMAC [16]. However, these algorithms have different ease of being attacked. We assume that the more dependable microservice uses stronger security techniques. Meanwhile, microservice may use multiple techniques to tackle security. For instance, a storage service may include access control, data replication, resource isolation, etc. Therefore, it is extremely desirable to judge the security levels of a microservice by exploiting and aggregating the adopted techniques.
- **[C3]** *Microservices are exposed to uncertain and unpredictable network environments.* In container management systems, the network is typically configured in a *best effort* manner [7]. The host machine tries to provision network resources to running containers. Consequently, the network performance interference drastically manifests outside the confines of strict QoS guarantee due to unpredictable *neighbors* [7]. This uncertainty even magnifies when microservices are geo-distributed across multiple Cloud datacenters through wide area Internet. In fact, network jittering with unpredictable latency is sometimes difficult to capture[8] [9].

Dependable Orchestration Requirements. Based on aforementioned analysis, the configuration diversity in terms of security capability and network uncertainty significantly affect the dependability of containerized microservices deployment, thereby reducing the satisfaction level of user's requirement. To achieve a dependable microservice orchestration, we need to deal with following problems: **[P1]** how to characterize new security risks associated with the microservices and how to bridge and exploit risks and security techniques to quantify the security goal satisfaction in microservice orchestration? **[P2]** how to capture the environmental impact of network QoS fluctuation (e.g. throughput and response latency) on microservice selection? **[P3]** How to deal with the increasing computation complexity of finding the most dependable microservices composition under different optimization constraints? The significantly increased scale of application workflow and increased number of microservices candidate horizontally and vertically boost the computing difficulty.

TABLE 1
Mapping between Cloud risks and security goals

Security Goal (G)	**Risk** (R)
Accountability	R1, R3, R5, R6, R7
Audibility	R1, R3, R6, R7, R8
Authenticity	R3, R4, R5, R8
Availability	R1, R2, R4, R7, R8, R9
Confidentiality	R1, R3, R4, R5, R7
Integrity	R1, R3, R4, R5, R7
Non-repudiation	R1, R7
Privacy	R1, R2, R3, R5, R6, R8

Problem Formal Definition. Formally an orchestrated application can be represented as a directed acyclic graph (DAG) $G = (S, E)$. Vertices S correspond to the service components and edges E correspond to data transferred between them. More specifically, each component i in the application has a group of candidate microservices S_i and a candidate p is denoted by $s_{ip} \in S_i$. We use s_{ip} and s_{jh} to represent the selected microservice of components S_i and S_j in the microservice workflow respectively. Accordingly, the dependency $E_{i,j}$ can be determined by $d_{s_{ip} \to s_{jh}} \in E_{i,j}$, where the data is transferred from the candidate p of S_i to the candidate h of S_j. To tackle above challenges, we leverage whitebox- and topology-dividing based method to quantify the achievable security of $G = (S, E)$. Each candidate s_{jh} might have varying data transmission rates and complicated patterns (e.g. latency between request and response) among different microservices at different times. We use statistically significant models to measure environmental uncertainties in a blackbox and statistic manner.

3 Quantitative Security Measurement

Security is one of the indispensable non-functional aspects for Cloud-based applications. Most research focuses on improving the security of each service component within applications. SecGraph for example is a technique that enables data owners to anonymize their data, while measuring the data's utility, and evaluating the data's vulnerability against modern De-Anonymization (DA) attacks [17]. However, there are a number of security techniques that are applied to a microservice. Therefore, how to measure the security level of each candidate microservice becomes difficult, while essential for composing a set of microservices to meet the security requirements.

To deal with **[P1]**, we first uncover the relationship between security level, goal, risks and the techniques/policies. A microservice may face some security risks due to various reasons such as deploying unreliable resource, using weak authentication techniques and being accessed by insecure

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2929389, IEEE Transactions on Parallel and Distributed Systems

devices etc. On the other hand, there are different security techniques or policies provisioned by the microservice to alleviate the corresponding risks, thereby enhancing the security. Therefore, the achievement of security goals of a microservice depends on both potential security risks and the mitigation degree of risk through techniques. We measure the security level of each microservice by the aggregation of targeted security goals. In next subsections, we investigate the relationships among the top risks within Cloud computing [18], the most popular security-enforcement techniques and Cloud security goals. We further develop a generic method to measure the security level of each microservice based on the multiple mapping relationships.

3.1 Mapping between Security Goals and Cloud Risks

Basic Idea. A complete collection of security goals within the CIA(Central Intelligence Agency) triad (including *accountability, audibility, authenticity, availability, confidentiality, integrity, non-repudiation and privacy*) has been illustrated by [19]. Cloud security is one of the subcategories within information security realms, thus we consider these goals as factors to quantify the security level of each candidate microservice. As discussed above, the achievement of security goals is affected by the potential security risks included in the microservice. Since microservice technology is still at an early but rapid development stage, there are limited standards or whitepaper of microservice security risk that can be widely accepted and generally referred to. Therefore, we currently applied the commonly-used cloud securities and risks into our main framework.

In the microservice scenario, the vulnerabilities can be leveraged by attackers more easily, thereby aggravating the manifestation of risks. The first reason for this is the architecture evolution. As monolithic application is de-composed into hundreds of smaller microservices, the number and complexity of interactions and communications between them is substantially increased. Attackers can thus exploit the decomposition and the inherent complexity to launch attacks against applications. For example, many separate APIs and ports per app represent numerous doors for intruders to try to access within an application. Such microservices may also be deployed in a cloud environment that the application owner is unable to control. Hence, the manifestation of specific risk such as loss of governance will increase accordingly. Secondly, container-based microservices provide attackers more chances to break the applications' security guard down. For example, unlike in a VM, the kernel is shared among all containers and host, which magnifies and disseminates the kernel vulnerabilities. If a container manages to attack the kernel, the corresponding threat can be spread into the other parts of the system, resulting in the whole host breakdown. Moreover, sharing and building container images tends to be much easier and more efficient than sharing VM images for DevOps purposes. This characteristic, however, increases the chances of privacy leaks, where the malicious code can be injected in both in-house written code and third-party libraries by the image owner [20] and image users are unaware of this.

The following investigates the top security risks in Cloud computing hinder the achievement of the listed security goals. These security risks have been summarized in [18]: **[R1]** Loss of governance, **[R2]** Lock-in, **[R3]** Isolation failure, **[R4]** Management interface compromise, **[R5]** Data protection, **[R6]** Insecure or incomplete data deletion, **[R7]** Malicious insider, **[R8]** Customers' security expectations and **[R9]** Availability chain. In order to obtain the correlated mapping between security goals and Cloud risks, we analyze all the potential attacks caused by each risk and inspect whether each attack will lead to unreachable security goals. The following example demonstrates how we map a risk to the corresponding security goals.

Example. Risk [R1] describes the case where a Cloud user loses the supervision to the Cloud provider in many aspects: lack of port scans, vulnerability assessment and penetration testing. The provider might outsource or subcontract services to third-parties (unknown vendors), resulting in uncertain security guarantees. Specifically, third-party providers might not be able to meet the organization's requirements of certifications and responsibilities, thereby violating the *Accountability*. Similarly, *Availability*, *Confidentiality*, *Integrity* and *Privacy* of data cannot be guaranteed for the same reason. Since no user audit is allowed by the Cloud provider, the risk will also be closely relevant to *Non-repudiation* and *Audibility*. Based on this domain knowledge, we can figure out the complete mapping as shown in Table 1.

3.2 Microservice Security Level

In this subsection, we use the installed risk prevention techniques of microservices to measure the security levels of them. Table 2 shows the notations of the rest of the paper. We assume that the security level of a microservice depends on the quantity and quality of the application of security techniques. For example, a type of risk can be prevented by many security techniques, and they have a variety of performance when they are applied independently. Moreover, we assume that there is incremental effect by applying more suitable techniques to a microservice. The following details how to calculate the security level of a microservice.

We assume that a risk R_k can be handled by a set of techniques and policies $Tech_{R_k}$. Thus, the total capacity of R_k that can be prevented is: $\sum_{j \in Tech_{R_k}} V_j$, where V_j represents the performance of technique j to reduce risk R_k. Next, if a microservice s offers a set of techniques and policies $Tech_s$, the ability of s to handle risk R_k can be represented as $R_k(s) = \frac{\sum_{i \in \{Tech_s \cap Tech_{R_k}\}} V_i}{\sum_{j \in Tech_{R_k}} V_j}$. As mentioned in Table 1, a security goal g^q corresponds to a group of risks R_{g^q}. Therefore s's capability of satisfying the security goal g^q can be defined as $g_s^q = \sum_{R_k \in R_{g^q}} R_k(s)$. We use set $G_s = \{g_s^1, .., g_s^{|G|}\}$ to represent s's capacity of targeting different security goals.

3.3 Microservice Security Measurement

Take the e-banking example in Figure 1 again. We are obliged to target security goals in essential microservices such as payment and customer information provider, etc. and enable secure data transmission (particularly user data and monetary information) between microservices.

Microservice Security Satisfaction. To identify and sort out those microservices that can best satisfy customers' security requirements, we extend *DSD (Degree of Security*

TABLE 2
Notations

Symbols	Descriptions
Security Metrics	
$g^q \in G$	The q^{th} goal where $\|G\| = 8$
$R_{g^q} \in R$	The q^{th} goal's risks set where $\|R\| = 9$
$R_k(s_i)$	The ability of s_i to handle risk R_k
$g^q_{s_i}$	s_i's capabilities of targeting security goal g^q
G_{s_i}	s_i's capabilities of targeting all goals
Microservice Orchestration	
u	A customer/user
s_{jh}	A candidate h selected by microservice component S_j
$u^{s_{jh}}$	u has selected s_{jh}
$d_{s_{ip} \rightarrow s_{jh}}$	The transfer of data from s_{ip} to s_{jh}
Security Measurement	
$DSSD(u, s_{jh})$	Degree of s_{jh} security deficiency, between its capability and u's requirements
$DDSD(d_{s_{ip}, s_{jh}})$	Degree of data security deficiency between two conjunct microservices s_{ip} and s_{jh}
$VDDSD(u, s_{jh})$	The security satisfactory degree of moving input data from other conjunct microservices to s_{jh} for user u
$VSEC(u, s_{jh})$	All security-related metric value of s_{jh} for user u
$Util$	The utility function for selecting a microservice
Network QoS Measurement	
$\varepsilon(t)$	The reduction degree of time consumed or transmission delays by using off-peak as the baseline
DTL	Data Transmission Latency
$SUR(t)$	Represents the possibility that a client can get the response from a Cloud service after t
$T(mrt, s_{jh})$	Customized user requirements for each service s_{jh}. If it meets requirements, set to 1 (available), else set to ∞
λ	Optimal solution for the given workflow
λ'	One of the solutions for the given workflow
Algorithms	
S	The set of all microservice candidates
S_i	The microservices candidate set for components i
M	The total population size of each generation
L	The numbers of components in a workflow
N	The number of Slaves
$Partition$	Partitions are basic units of parallelism and each partition is responsible for an atomic logical division of data.
Q'	The partition number used for Fitness Calculation
Q''	The partition number used for Genetic Operation
C	The number of cores on each Slave
T	The number T of CPUs requested by each task
R	The threshold for the number of stable iterations in termination
$iter$	The total number of GA iterations
C_i	The i^{th} constant value
IN_{S_j}	The inputs from S_j's immediate predecessors
IN	A set of all data dependencies in the given workflow
Algorithms Analysis	
E	The size of the elitism list
$dCollect_i$	The time delay for collecting the local elitism list from i^{th} partition
$dBroadcast_{E_i}$	The time delay for broadcasting the global elitism list to i^{th} Slave

Deficiency) [21] into *DSSD (Degree of Service Security Deficiency)* in our model to describe the discrepancy between desired security level and the supplied level. In this context, $DSSD(u, s_{jh}) = 0$ if the selected microservice s_{jh} can fully meet the demands of customer (u). DSSD can be formalized as:

$$DSSD(u, s_{jh}) = \sum_{q=1}^{|G|} w^q_j \cdot M(g^q_u, g^q_{s_{jh}})$$
$$\text{where } 0 \leq w^q_j \leq 1,\ \sum_{q=1}^{|G|} w^q_j = 1 \text{ and} \tag{1}$$
$$M(g^q_u, g^q_{s_{jh}}) = \begin{cases} 0 & \text{if } g^q_{s_{jh}} \geq g^q_u \\ g^q_u - g^q_{s_{jh}} & \text{otherwise} \end{cases}$$

where w^q_j is the weight of the q^{th} security goal for component S_j and the customer specifies the weights to reflect their priorities among those goals. M represents the disparity between s_{jh} security levels and u's security demands in terms of each security goal.

Data Transmission Security Satisfaction. For data transmission between two components S_i and S_j, we use *DDSD (Degree of Data Security Deficiency)* to depict the satisfaction of security of the data movement:

$$DDSD(d_{s_{ip} \rightarrow s_{jh}}) = \sum_{q=1}^{|G|} w^q_{ij} \cdot M(g^q_{s_{ip}}, g^q_{s_{jh}})$$
$$\text{where } 0 \leq w^q_{ij} \leq 1,\ \sum_{q=1}^{|G|} w^q_{ij} = 1 \text{ and} \tag{2}$$
$$M(g^q_{s_{ip}}, g^q_{s_{jh}}) = \begin{cases} 0 & \text{if } g^q_{s_{jh}} \geq g^q_{s_{ip}} \\ g^q_{s_{ip}} - g^q_{s_{jh}} & \text{otherwise} \end{cases}$$

In Eq. 2, $d_{s_{ip} \rightarrow s_{jh}}$ indicates the data transfer from source s_{ip} to the destination s_{jh}. We assume that the candidate microservice in component S_i satisfies the expected security level of customers. Therefore, $M(g^q_{s_{ip}}, g^q_{s_{jh}})$ directly indicates the customer's security satisfaction from s_{ip} to s_{jh}. Customers can also define the priority or importance of each security goal when transferring data from S_i to S_j by adjusting w^q_{ij}. Thereafter, we can estimate the satisfaction of transferring holistic data:

$$VDDSD(u, s_{jh}) = \sum_{\{d_{s_{ip} \rightarrow s_{jh}}\} \in IN_{S_j}} DDSD(d_{s_{ip} \rightarrow s_{jh}}) \tag{3}$$

where IN_{S_j} is all input data pipelines of S_j from all its immediate predecessors. Finally, we can calculate s_{jh}'s satisfaction of security by Eq. 4:

$$VSEC(u, s_{jh}) = VDDSD(u, s_{jh}) + DSSD(u, s_{jh}) \tag{4}$$

4 Environmental Uncertainty Measurement

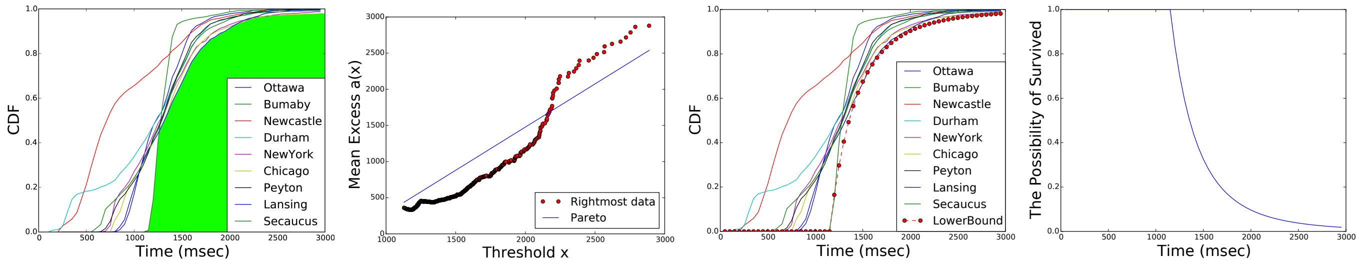
To solve **[P2]**, we leverage a data-driven and statistical modeling method to capture and describe the environmental impact on the dependable microservice orchestration.

4.1 Data Transmission Latency

Due to the network bandwidth fluctuation, the data transmission latency varies between upstream and downstream microservices. The throughput of file transfer to run instances within an Amazon EC2 datacenter (US East Northern Virginia Region) is evaluated in [22]. The experiment captured the network bandwidth every 30 minutes between 20th and 21st May 2013. A throughput surge can be detected during the period 7.00-8.00 a.m. while the throughput during other periods is much less and remains stable. Therefore, we can take 7.00-8.00 am as the *peak time* and distinguish it from other off-peak times. We define a latency factor ε,

$$\varepsilon(t) = \begin{cases} \frac{mean(thr(\bar{t}))}{mean(thr(t))} & \text{if } t = \text{peak time} \\ 1 & \text{if } t = \text{off-peak} \end{cases} \tag{5}$$

where $thr(\bar{t})$ indicates the throughput of time periods $\bar{t}$ (time period except for t). According to real data statistics [22], the peak throughput is approximately twice that of off-peak throughput, resulting in $\varepsilon = \frac{1}{2}$. In practice, ε is configurable and can be adjusted according to different conditions. Moreover, the total time and resource consumption also rely



(a) (b) (c) (d)

Fig. 3. Distributions; (a) RTT from different locations (b) ME of lower bound data and the fitted distribution (c) The lower bound of the end to end network delay in Cloud datacenter (d) The survival distribution of the measured datacenter

on the amount of data. We can thus define the overall *DTL (Data Transmission Latency)*. $Data_{S_j}$ and $Total$ are the input size of component S_j and the overall data consumed within the entire microservice workflow respectively. The given time $t_{s_{jh}}$ represents the local time of s_{jh}.

$$DTL(t_{s_{jh}}) = \varepsilon(t_{s_{jh}}) \cdot \frac{Data_{S_j}}{Total} \tag{6}$$

Note that the size of transferred data and the execution time of each microserivce can be taken from provenance logs. Numerous works [23][24] describe how to estimate the time and size based on the execution logs.

4.2 Uncertainty Modeling

4.2.1 Data-driven Benchmarking and Observations

In order to elaborate on the uncertainty model and its impact on service provisioning efficiency and availability, we probe the request response latencies and regard them as a main indicator. To explore a generic distribution of the request response time of a specific location, we deploy the same implementation of clients that are geographically distributed around the world, and send requests to one central datacenter (in Dublin). The round trip timespan (RTT) is collected, and the experiments are repeated 1,400 times for each individual client within a continuous 24-hour period. As shown in Fig. 3(a), requests from different cities are sent to Dublin and we display the response time with *CDF (Cumulative Distributed Function)*. Apparently, distance and location have a significant impact on the distribution. In general, a longer distance indicates a higher probability of traffic delay. In addition to these observations, we have to form a uniform model and find a marginal distribution *to cover all distributions of different requests comprehensively, and satisfy the worst case* for each specific datacenter. In Fig. 3(a), the target distribution is supposed to be within the colored area, and it is highly desirable to obtain an optimal distribution that fits the marginal bound as closely as possible. Due to space limitations, we only briefly demonstrate how to find the required distribution.

4.2.2 Mathematical Modeling

Basically, the exponential distribution class provides a rigorous mathematical framework. In reality, our target distribution belongs to a subclass of exponential distribution – *heavy-tailed* distribution [25]. We intend to find a regularly varying tail and determine the most appropriate distribution with the relevant properties.

Firstly assume a non-negative random variable X, the CDF of X is $F(x) = P[X \leq x](x \geq 0)$ and $\bar{F}(x) = 1 - F(x)$. The data (shown in Fig. 3(a)) conforms to Eq. 7, and thus the distribution is subexponential (i.e. $F \in \mathcal{S}$) [26].

$$\lim_{x\to\infty} \frac{P[X_1 + \cdots + X_n > x]}{P[max(X_1, \ldots, X_n) > x]} = 1 \quad \textit{for some (all)}\ n \geq 2, \tag{7}$$

where $X_1, \ldots, X_n$ are *independent and identically distributed* random variables (IID). [27] further substantiates the implication that such F is regularly varying-tailed (heavy-tailed). The heavy-tail distribution will have:

$$\lim_{x\to\infty} \frac{\bar{F}(x)}{e^{-\delta\cdot x}} \to \infty \quad \forall\delta > 0, F \in \mathcal{S} \tag{8}$$

In fact, this indicates that the tail of F decreases more slowly than any exponential tail. Meanwhile, for a positive measurable function f which regularly varies with index α, if Eq. 9 is satisfied, we can write $f \in \mathcal{R}(\alpha)$ [28]:

$$\lim_{x\to\infty} \frac{f(t\cdot x)}{f(x)} = t^{\alpha} \quad \forall t > 0 \tag{9}$$

Due to the heavy-tailed pattern, we conclude $\bar{F}(x) \in \mathcal{R}(-\alpha)$ and further represent $\bar{F}(x) = x^{-\alpha} \cdot l(x)$ where $x > 0$, $\alpha > 0$, and $l(x) \in \mathcal{R}(0)$ is a slowly varying function. In general, Pareto, Brr and Log-gamma distributions are examples of distribution functions with such regularly varying tails.

Afterwards, we have to determine which offers the best-fit distribution. To this end, we use the *mean-excess (ME)* metric (see Eq. 10) to sort out the desired distribution from all these candidtates [29].

$$ME(x) = E(X - x|X > x) = \frac{\int_x^{\infty} \bar{F}(y)dy}{\bar{F}(x)}, \quad x > 0 \tag{10}$$

We extract the data which corresponds to the lower bound in Fig. 3(a), and calculate the ME value with different x ranges as shown in Fig. 3(b). We examine the ME function of each distribution, and only the Pareto Type I distribution increases linearly [30]. Specifically, the ME of a standard Pareto I distribution is shown in Eq. 11 (the blue line in Fig. 3(b)).

$$ME(x) = \frac{x}{\alpha - 1}, \quad \alpha > 1 \tag{11}$$

Therefore, we choose Pareto as our approximate fitting distribution and use Eq. 12 to calculate the cumulative probability within a specified time window.

$$F(t) = \int_t^{\infty} \frac{\alpha \cdot x^{\alpha}}{t^{\alpha+1}} dt \tag{12}$$

Fig. 3(c) illustrates the effectiveness of the application with Eq. 12. The *LowerBound* curve can always guarantee the coverage of the response time boundary and all measured request cases. In this manner, we have the *survive function* (Eq. 13), which represents the possibility of a client getting a response from a Cloud service after t (as illustrated in Fig. 3(d)). By leveraging the survive function, we quantitatively describe the uncertainty in request handling.

$$\bar{F}(t) = 1 - F(t) = 1 - \int_{t}^{\infty} \frac{\alpha \cdot x^{\alpha}}{t^{\alpha+1}} dt \tag{13}$$

5 Formulation and Solution

5.1 Orchestration Problem

Based on the model proposed in Section 3.3 and Section 4, the overall function of user u to run a microservice candidate h in the component S_j can be expressed as:

$$Util(u, s_{jh}, t) = \theta \cdot\ DTL(t_{s_{jh}}) + (1 - \theta) \cdot VSEC(u, s_{jh}) \tag{14}$$

Users can customize the weight θ to manifest their required importance and tradeoff between data transmission efficiency and the security requirements.

Single component utility function. For each component S_j, customers have their specific requirements to avoid the uncertainty. Based on our early experience, we assume that the survival rate of S_j ought to be higher than 10%. In order to apply the uncertainty factor into our model, we depict $T(mrt{,}s_{jh})$ as in Eq. 15 where mrt is the maximum response time that the customer can tolerate. In this formalized definition, if the survival rate is lower than 10%, we set the output to be $+\infty$.

$$T(mrt, s_{jh}) = \begin{cases} 1 & \text{if } \bar{F}(mrt) \geq 10\% \\ +\infty & \text{otherwise} \end{cases} \tag{15}$$

Thus, the utility of u selecting an s_{jh} is:

$$Value(u, s_{jh}, mrt, t) = T(mrt, s_{jh}) \cdot Util(u, s_{jh}, t) \tag{16}$$

where $s_{jh} \in S_j$ and $S_j \in S$.

Multiple component orchestration. We assume that the optimal orchestration is that each selected microservice s_{jh} in microservice component S_j within an application can meet user u's requirements. It can be defined as:

$$Optimal(u, \lambda) = \sum_{s_{jh} \in \lambda} Value(u, s_{jh}, mrt, t) = 0 \tag{17}$$

where λ is one of the optimal solutions for the target orchestration workflow. However, it is very difficult to guarantee the optimal solution. The optimal solution may not even exist due to the lack of candidate services. Therefore, our objective is to find an orchestration that minimizes the value from all possible solutions Λ:

$$\textbf{Minimize:} \quad \sum_{s_{jh} \in \lambda', \lambda' \in \Lambda} Value(u, s_{jh}, mrt, t); \tag{18}$$

where λ' is one of the solutions for the target workflow.

Definition 1 (*Optimal Solution*)**.** As in the statement of optimization problem in Eq. 18, the number of selected microservices is fixed to $|\lambda'|$. Therefore the optimal solution can be obtained by traversing $m^{|\lambda'|}$ solutions, where each component in the orchestration workflow has m candidate microservices. The given optimization problem can be proofed an NP-hard problem.

Conjecture 1. *The dynamic programming method is not suitable for solving our problem.*

We assume that a set $S_K = \{s_1, ..., s_k\}$ is the optimal solution of composing k types of microservice, and a set S_{K+1} is the optimal solution of composing $k+1$ types of microservices. If $S_k \not\subset S_{K+1}$, the dynamic programming method will take more time than the direct calculation of the optimal S_{K+1}. However, in our optimization problem, this situation happens very frequently due to the impact of data dependencies. Thus, a new and more efficient algorithm is desired to solve this optimization problem.

5.2 Basic Solution Principle

With respect to optimization algorithms, [13] demonstrated that Genetic Algorithms (GA) outperform other solutions such as Mixed-Integer Non-linear Programming (MINLP) or Linear Programming (MIP), etc. The high time complexity of MIP and MINLP solvers make it infeasible to apply those in large orchestration problems. Therefore, we present GA-Par (Genetic Algorithm Parallelism) – a novel Parallel Genetic Algorithm designed to optimize the microservice orchestration in terms of the security under uncertainty.

In our Genetic Algorithm (GA), a string $w = (\beta_1, ..., \beta_L)$ with length $L = |\lambda'|$ is used as a chromosomal representation for a workflow and each $\beta_i = S_i \in S$ represents a component. Each gene segment derives from a finite set of microservices $S_i = \{s_{i1}, ..., s_{im}\}$. According to our optimization objective, we set the fitness function $fit(\lambda') = \sum_{s_{jh} \in \lambda'} Value(u, s_{jh}, mrt, t)$. Basically, the GA usually includes two primary parts: *Fitness Calculation* (FC) and *Genetic Operations* (GO) [31]. The effect value of each chromosome (individual) is calculated in FC, and then the elites are sorted out from all population members. In GO, the new chromosomes are generated by operations: *Selection*, *Crossover* and *Mutation*. After iterations of several generations, a sub-optimal solution can be found.

6 GA-Par: Design and Implementation

To tackle **[P3]**, we propose a novel multi-phase parallelized GA to accelerate the orchestration procedure. We will briefly overview existing solutions and introduce our architecture and detailed implementations.

6.1 Parallelized GA Overview

According to the philosophy of divide-and-conquer, GA parallelism can be categorized into two approaches:

1) Fitness Evaluation Parallelism – The parallelism is conducted merely within the calculation of fitness value by distributing individuals to different computing nodes (i.e, slaves). The centralized master manages all the GO operations and the generated population. This method is suitable for cases that have a time-consuming fitness function calculation. Obviously, the GO will become extremely inefficient with an increase in population scale due to the single point of performance bottleneck. The computing power of the master tends to be limited once dealing with the huge number of individuals for generating new populations. Moreover, a significant network latency might be aggravated considering the communication of individual distribution when computing the fitness value over different slaves.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2929389, IEEE Transactions on Parallel and Distributed Systems

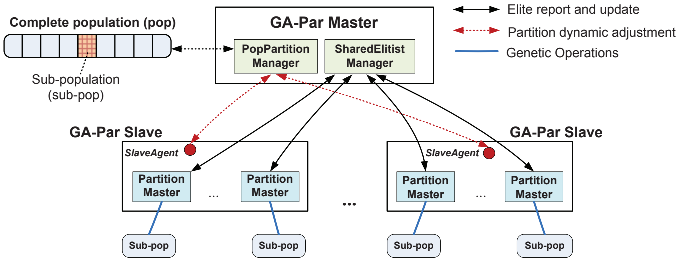


Fig. 4. The architecture and module interactions in GA-Par

2) Population Reproduction Parallelism – This approach allows the process of creating a new population independently over different slaves, and each slave generates and manages a *subset* of the new population. As a result, the time consumed in the population generation can be reduced. However, if the GO is only performed inside each isolated computing node, the probability of duplicated individuals will rise. Thus, it is extremely likely to fall into a local optimum.

Recently, [32] introduced a framework for running genetic algorithms with map reduce paradigm in Hadoop. However, the design is on the basis of dumping each generation of chromosomes onto disk, thereby dramatically introducing huge IO costs and decreasing the operational performance. Furthermore, a large scale workflow optimization problem requires a very large search space in terms of horizontal (dependencies of each components) and vertical (different attributes of each candidate microservice). For these reasons, in-memory data operations naturally fit in the GA operations, especially for the crossover operation in which the data for each individual can be efficiently shuffled among different nodes. We therefore propose a new parallel algorithm GA-Par based on Apache Spark and it adaptively incorporates the parallelization of both fitness evaluation and population reproduction.

6.2 GA-Par Framework

6.2.1 Architecture

GA-Par adopts the master-slave architecture and two-phase parallelization management to fully explore the efficiency of the fitness calculation and genetic operations. The parallel degree can be dynamically adjusted in order to maximize the calculation velocity whilst maintaining the optimization quality. Mechanisms such as adaptive partition control, global population shuffling and elitist sharing are proposed to achieve these objectives.

Fig. 4 details the architecture and multiple modules underpinning the implementation of GA-Par. The GA-Par Master is responsible for the overall management of parallelization and population life-cycle (i.e. generation and exchange). The SlaveAgent is the daemon agent that runs on each GA-Par slave node and handles instructive messages from the GA-Par Master. More specifically, the population will be divided into a number of sub-populations and each of them is firstly initiated on each partition. This procedure is controlled by the PopPartitionManager (PPM) with the master. Correspondingly, on each slave node, the PartitionManager (PM) takes charge of each sub-population and manages the new generation through FC and GO.

Algorithm 1: : GA-Par Master

```
Input:
  $S$: all candidate microservices
  $Q'$: partition number for FC
  $Q''$: partition number for GO
  $\lambda'$: required composition of microservices
Master($S, Q', Q'', \lambda'$)
  $sharedElitist \leftarrow$ Initialize($S, \lambda'$)
  while do not meet termination condition do
    setPartitionNumberToSlaves($Q'$)
    // wait all PartitionMasters to finish FC
    sync for $FC$
    // collect all elites from partitions to a list in master node
    $localElitistList \leftarrow$ collect()
    // find the best individual
    $s \leftarrow$ findBest($localElilistList$)
    // update the shared elitist list
    $sharedElitist \leftarrow$ update($sharedElitist$,
      $localElilistList$)
    // broadcast the shared elitist list to slaves
    broadcast($sharedElitist$)
    setPartitionNumberToSlaves($Q''$)
    // wait all PartitionMasters to produce new generation
    sync for $GO$
  end
  return $s$
```

Also, the PPM coordinates the dynamic partition configurations to make partitions fully parallelized over different slaves whilst considering the data shuffling cost (Appendix A.1 explains the execution model). To increase the diversity of the new generation, we merge several partitions into a single partition for population reproduction and then repartition it appropriately to ensure the parallelization of FC (detailed later). Moreover, the local elite results of each partition will be collected and synthesized into a global elitist list by SharedElitistManager(SEM) in GA-Par Master before the selection phase. The manager will then broadcast the global elitist list to the running partitions to ensure each of them has the best chance to generate advantage offsprings.

We adopt Spark to implement the aforementioned functionalities. Compared with Hadoop, Spark can provision a more flexible approach to manipulate data via Resilient Distributed Datasets (RDD), which a collection of elements that will be partitioned across different slaves [33]. Moreover, the in-memory computation on large clusters strikes the balance between the maximized parallelism and the diversity control.

6.2.2 Two-phase Parallelization

A candidate microservice within a workflow component must satisfy all security and efficiency requirements, while guaranteeing service availability. Herein, we describe the asynchronized messaging between GA-Par Master and Partition Managers and the main procedures on both sides realize the adaptive partition operations. GA-Par Master will finally output the optimal orchestration solution.

Alg. 1 and Alg. 2 depict the collaborative and interactive procedure of the master and pertaining slaves. The master will read candidate list, configurations before initializing the shared elitist information. Regarding the slave, after receiving the candidates from the GA-Par Master, individuals will be initialized on Q partitions, followed by the GO within each partition. The population size per partition is $\frac{M}{Q}$, where M is the total population size of each generation, which has been pre-defined. Q is the number of partitions selected from the set $\{Q'', Q'\}$, where $Q'' < Q'$. Q can therefore be dynamically adjusted in different execution phases to ensure the tradeoff between parallelism and diversity. In particular, during the FC phase, in order to rapidly finish the FC computation and obtain the elitist list, the degree of parallelism should be maximized to Q'. In other words, the individuals of the whole population (M size) are distributed to the maximal number of partitions, in order to fully utilize the computational resources. On the other hand, in the GO phase, individuals within the same partition are involved to produce a new generation. Therefore, the more individuals involved, the more diverse generations can be produced. As a result, there are increasing chances to obtain individual advantage. Accordingly, as shown in Alg. 1 (Lines 4 and 15) and Alg. 2 (Lines 6 and 13), the partition number is shrunk from Q' to Q'' to guarantee the offsprings' diversity.

It is noteworthy that the elitist list is computed by the extension method from [31] to guarantee the diversity of each generation. The local elitist list will be sorted out, sent within the PartitionMaster (Alg. 2 Line 10-12) and collected in the GA-Par Master (Alg. 1 Line 8). The local list will be aggregated into a $sharedElitist$ by selecting the individuals that have better fitness values. Afterwards, the global $sharedElitist$ will be disseminated to each PartitionMaster across different slaves (Alg. 1 Line 9-11). Compared with conventional methods, our elitism method also benefits from the global elitist list over the partitions, giving rise to the fact that the outstanding individuals of each partition can be selected and synchronized. This can significantly minimize the fitness value and reduce the execution time. The PartitionMaster continues the GO phase by using the latest synchronized list. When new generations are generated and notified from the slave (Alg. 2 Line 18 to 20), GA-Par Master will update by selecting the individuals that have better fitness values (Alg. 1 Line 17) and repeat the iteration loop until a certain condition emerges. Finally, GA-Par Master outputs the best fit microservice selection for the submitted workflow (Alg. 1 Line 19) .

6.2.3 Algorithm Termination

The GA algorithm will eventually compute the best solution as the total number of iterations $iter$ goes to ∞ [34]. However, limited by computation resources, the value of $iter$ should be decided in some sense of *optimal*. The termination control in GA-Par assumes that the process will be terminated if there is no further improvement in the global fitness value for a fixed number of iterations R. Literally, the value of R can be defined by users according to the complexity of their problems. In our experiments, we perform the grid search to find the optimal R within the range of [10, 20, 30, 40]. Eventually 20 is selected since we cannot observe any efficiency gain of reaching the optimal when R surpasses 20.

Algorithm 2: : GA-Par Slave and PartitionMaster

```
   Input:
   $shareElitist$: shared elitist list
   $Q'$: partition number for FC
   $Q''$: partition number for GO
   $pop$: population
   $M$: the population size
1  Slave($shareElitist$, $Q'$, $Q''$, $M$, $pop$)
2      // initialize a $\frac{M}{Q'}$ size population for each partition
3      $pop \leftarrow$ Initialize($M, pop$)
4      while not be terminated by master do
5          // get the partition number $Q'$ from Master
6          PartitionNumber($Q'$)
7          // compute the fitness value for each individual
8          $fitpop \leftarrow$ FC($pop$)
9          // find the best $N$ individual
10         $localElitist \leftarrow$ find($fitpop, N, pop$)
11         // send $localElitist$ to Master
12         Send( $localElitist$)
13         // sync latest $shareElitist$ from Master
14         $shareElitist \leftarrow$ Sync()
15         // get the partition number $Q''$ from Master
16         PartitionNumber($Q''$)
17         // generate a new generation
18         $pop \leftarrow$ GO($pop$, $sharedElitist$)
19         // notify new generation pop to Master
20         Notify()
21     end
```

6.3 Time Complexity Analysis

Since our algorithm involves intensive computation while having low I/O consumption, we simplify our analysis by only providing the majority cost for one iteration (generation) under the case of a single core executor: 1) population creation/reproduction: $\mathcal{O}(M|S|)$; 2) fitness score computation: $\mathcal{O}(M(L^3))$; 3) elitism list updating: $\mathcal{O}(\mathcal{O}(Mlog(\frac{M}{Q'}) + EQ'log(EQ'))$ where E is the size of the elitism list; 4) crossover: $\mathcal{O}(ML)$; and 5) mutation: $\mathcal{O}(M)$. The total complexity of one iteration for the single core executor is $\mathcal{O}(M(|S| + L^3 + log(\frac{M}{Q'})))$.

In contrast, under the parallel cluster environment, the time complexity of GA-Par computation can be reduced to $\mathcal{O}(\frac{T}{NC} \times [M(|S| + L^3 + log(\frac{M}{Q'})) + EQ'log(EQ')])$ where T is the requested CPU cores of each task, and C and N represent the number of CPU cores on each node and node number respectively. The analysis has been omitted and details are in Appendix A.

7 Experimental Evaluation

7.1 Experimental Setup

Platform. We perform all experiments on a 20-node cluster hosted on Google Cloud Platform. Each node is hosted as a n1-standard-8 instance with 8 vCPU (single hardware hyper-thread and chosen from 2.5 GHz Intel Xeon E5 v2, 30GB RAM, and 100GB storage.

Dataset. The used datasets consist of: 1) a list of $4,532$ web services' response-times distributed in 150 computing nodes across over 20 countries [35][1]; 2) one week throughput records that are derived from Dublin Microsoft Cloud servers [22]; 3) 51 collected techniques and policies for security improvements of Cloud-based services[2].

Workloads. In the following experiments, we assume that the customers prefer the highest security that Cloud-based microservices can provide, given by $g_{c_j}^q = 1, q \in \{1, .., 8\}$. To evaluate our algorithm, we firstly assign a number from 0 to 24 to each microservice, representing the local time period. Next, we randomly select n techniques out of total 51 techniques to each microservice and n is also randomly generated. In addition, we randomly generate four types of DAGs as workflows: The number of vertices in the DAG #w is selected from (50, 100, 150 or 200) and dependencies between vertices are randomly generated to simulate the workflow (e.g., the example in Figure 1). For each vertex within a workflow, we then randomly select #s = (100, 300, 500, 1000 or 1500) microservices as the candidates. We use different mappings of workflow and service configurations for the application (marked as *App(w, s)*) in our experiments. We deploy *App(w, s)* and set up the same initial population.

Methodology. We firstly evaluate the security enhancement by comparing GA-Par with the **SU** (a generalized security-unaware method) and **SA-Greedy** (a greedy-based security-aware approach). Afterwards, the overall orchestration effectiveness and efficiency of our GA-Par (multiple slaves with two-phase parallelization) will be evaluated against other schemes: **SGA** (a standalone GA, running GA in a single machine) and **HGA** (a Hadoop-based Parallelized GA, following the elephant56 [32]). Since the setting-up of mapper and reducer in [32] is equivalent to our partition setting during the two-phase parallelization, we evaluate GA-Par and HGA under completely the same configurations for fairness considerations. Furthermore, we conduct the analysis of impact on the GA-Par efficiency by varying parameters such as diverse application scale, parallelization setting, etc. The scalability of the proposed method is also assessed by varying the number of machines.

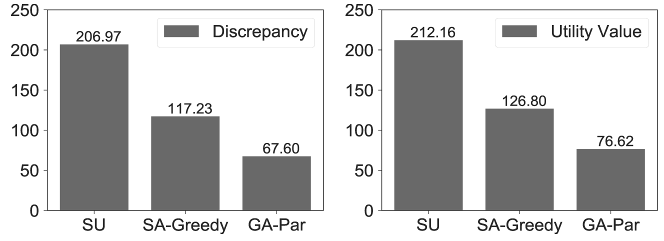
Metrics. In particular, we use three complementary metrics:

- **Security Discrepancy.** The discrepancy between the requirement and the targeted security level that the given approach can achieve.
- **Consumed Time.** The time consumption of the orchestration by different methods.
- **Utility Value.** The generated utility value of Eq. 18 by applying different algorithms. The value implies the reachable distance to the theoretically optimal solution.

7.2 Effect of Using Security-Aware Approach

In this section, we investigate the security improvement introduced by the security-aware mechanism by running the *App(100, 100)*. Basically, we evaluate 1) the presented security discrepancy between the user requirements and the actual provision by the selected service and 2) the achievable utility value by using different schemes.

As shown in Figure 5, the resultant discrepancy can be significantly reduced by 67.34% and 42.34% against SU and SA-Greedy respectively. This substantial decrease within GA-Par is because the proposed algorithm is aware of the security satisfaction in both services themselves and the data transmission. Additionally, an approximately 63% and 40% reduction of utility value can be observed compared with SU and SA-Greedy respectively. This is because only the network uncertainty (Eq. 6) is considered in the utility function when adopting SU, while our approach takes both uncertainty and security into account as shown in Equation 14. Although the uncertainty and security are included in SA-Greedy, the greedy algorithm only has one shot to compute the optimal solution, which makes it very hard to have a correct design to find the optimal solution. Moreover, this type of algorithm is limited in generalization in that each single optimization requires a new and carefully designed algorithm. The results indicate that holistically GA-Par can minimize the utility values and effectively find a more reasonable solution from the candidate space.



(a) Security Discrepancy (b) Utility Value

Fig. 5. Security Enhancement Effects by Different Methods

TABLE 3
Overall Evaluation by Submitting *App(100, 100)*

Exp Group	Time Ratio	Value Ratio
SGA/GA-Par	5.85	1.85
HGA/GA-Par	2.54	1.39

7.3 Overall Effect Evaluation

In this experiment, we firstly deploy an application *App(100, 100)* by using the SGA, HGA and GA-Par. The initial population size in all cases is uniformly set to be $5,000$.

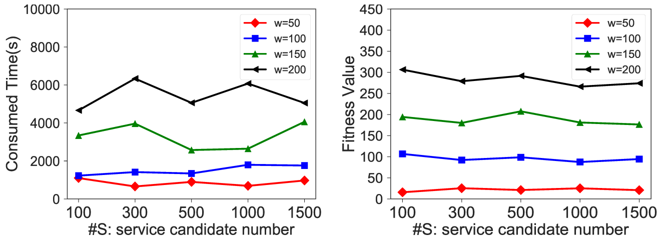
SGA vs GA-Par. It is observable from Table 3 that GA-Par obtains a better solution for the optimization problem in terms of fitness(utility) values, compared to SGA (i.e. the ratio between them is 1.85). Moreover, the time consumption of SGA is approximately 5.85 times more than that of GA-Par which is far beyond the range of tolerance. The reason is that the stage of reproduction requires a great

TABLE 4
Time and Value Ratio of HGA/GA-Par under Different *App(w,s)*

Time Ratio Comparison						
w \s	**100**	**300**	**500**	**1000**	**1500**	**avg**
50	3.901	2.844	3.407	4.789	4.132	*3.815*
100	2.182	2.535	2.670	2.684	2.691	*2.553*
150	2.415	4.065	2.391	2.571	3.775	*3.043*
200	3.277	3.965	4.544	3.720	4.416	*3.985*
avg	*2.944*	*3.352*	*3.253*	*3.441*	*3.754*	3.349
Value Ratio Comparison						
50	1.648	2.114	2.118	1.726	1.745	*1.870*
100	1.506	1.389	1.326	1.347	1.374	*1.388*
150	1.345	1.240	1.348	1.317	1.385	*1.327*
200	1.361	1.197	1.253	1.260	1.249	*1.264*
avg	*1.465*	*1.485*	*1.511*	*1.413*	1.438	1.462

[1] The original dataset is not available now, we therefore publish the dataset used in our experiments in [36]

[2] The list of the techniques and policies is available in [36].



(a) Time of GAPar (b) Value of GAPar

Fig. 6. Time and Value of GAPar.

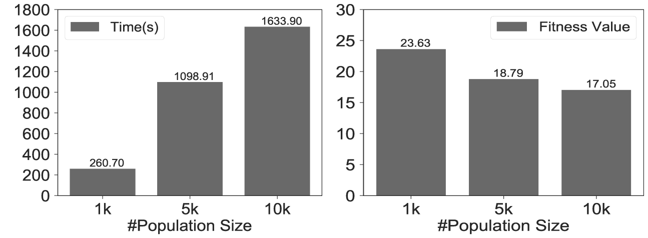
number of computing resources and the capacity of a single computation node typically cannot satisfy this vast demand. In contrast, by using two-phase parallelization, GA-Par can accelerate the procedures of both fitness value calculation and generation reproduction. Considering the intolerable time inefficiency, we will not take SGA into account in the following experiments.

HGA vs GA-Par. Table 3 demonstrates the ratios of time and value are roughly 2.54 and 1.39 for *App(100, 100)*, indicating that GA-Par outperforms HGA with respects to both time efficiency and the fitness calculation. Table 4 shows more detailed comparison results when deploying applications with different configurations of w and s. It is observable that the average time and value ratio are 3.349 and 1.462 respectively. There is a stable behavioral ratio between HGA and GA-Par in spite of some marginal fluctuations. Although the value ratio slightly decreases with the increment of workflow size, there still exists a guaranteed improvement of utility value by GA-Par. The improvements are predominately caused by the in-memory processing and communication and the efficient parallelization level within GA-Par. In effect, GA-Par obtains a trade-off for efficiency between fitness and time, as elite list sharing and synchronization guarantee the fitness efficiency and population division ensures the time efficiency. Furthermore, we will illustrate the significant reduction of consumed time and the improvement of effectiveness by varying parameters (such as Q, Q'') in GA-Par in following subsections.

7.4 Impact of Different Application Scales

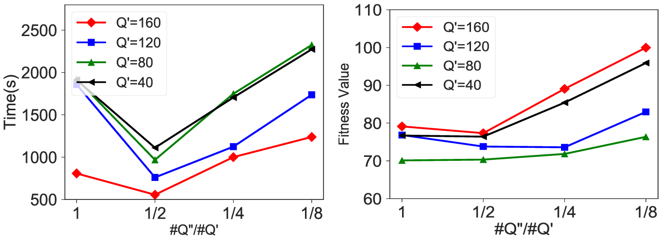
Figure 6 demonstrates the experimental results under different workflow size and candidate number of microservices by using GA-Par. We can observe that with the increment of workflow size, the time consumption increases accordingly. The linear increase demonstrates that the growth of task numbers in the workflow will increase the search range to find optimal solution, thereby taking longer time to finish the overall computation. In Figure 6(a), the number of service candidates is not an obvious factor that influences the time consumption. The consumed time slightly fluctuates when the topology and size of the workflow is determined. Apparently, given the workflow size w and each component in the orchestrated workflow has s candidates, the search space is s^w. Thus, the impact of s on the consumed time will not be as significant as that of w.

Likewise, a similar phenomenon can be observed in terms of the fitness calculation. In particular, the increased workflow size will naturally degrade the optimization effectiveness given the fixed setting of the total population. Compared with a smaller-scale workflow, larger workflows with soaring component numbers are less likely to converge and obtain the optimal result once the population is set up.



(a) Consumed Time (b) Fitness Value

Fig. 7. The impact of population size setting



(a) Consumed Time Comparison (b) Fitness Value Comparison

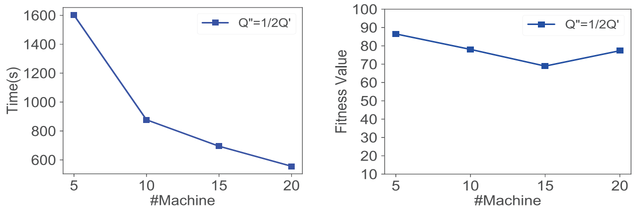
Fig. 8. Impact analysis of partition configuration

7.5 Impact of Parallelization Parameters

Herein, we discuss the impact of some parallelization parameters on the execution performance. We will evaluate the impact on the fitness value and execution time under different population size and partition numbers during the two-phase parallelization depicted in Section 6.2.2.

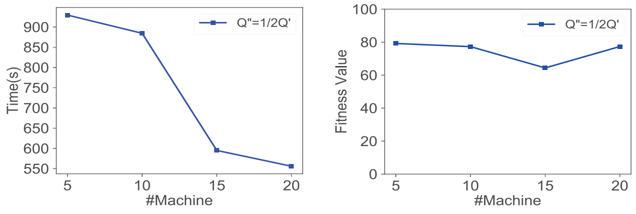
Firstly, we fix $App(50, 100)$ and change the initial population size ranging from 1000 to $5,000$ and $10,000$; and it is observable from Fig. 7(a) that the consumed time will dramatically increase with the population size going up while the fitness value reduced when the population size soars, as illustrated in Fig. 7(b). This is because the expanded population will increase the opportunities of emerged elitist since more individuals are involved in the genetic operations. Undoubtedly, it leads to increased effectiveness towards the optimal result, but with an expected extension of time consumption. Therefore, within the orchestration we need to strike the balance between the time consumption and orchestration quality.

Secondly, we fix $App(100, 100)$ with the $5,000$ population and dynamically adjust Q' and Q''. Q' ranges from 40 to 160 with an equal interval while Q'' is initially set to the same as Q' and will shrink to $1/2$, $1/4$ or $1/8$ of Q'. Figure 8(a) illustrates that with the decrement of Q', the time consumption grows accordingly. It is obvious that the increase of parallelization will accelerate the procedure of fitness calculation, where the computation time difference becomes more significant when Q' is large, while for small Q', the benefit of parallelism is trivial and can even be governed by the randomness introduced by our holistic algorithm. Additionally, given Q' is fixed, the execution time reaches the bottom when the partition number in the GO phase reduces by 50%. In fact, the resource capacity in our Spark-based computation model will determine the maximal number of co-locating executors within the same slave node, and the resultant merge scheme considering IO



(a) Time (b) Value

Fig. 9. Fixed number of partitions per node



(a) Time (b) Value

Fig. 10. Fixed number of total partitions

consumption and messaging communications. Specifically, if three vcores are requested per executor and the physical capacity is eight vcores per slave node, at most two executors can be simultaneously launched within a node during the fitness calculation phase. Equivalently, when $Q'' = 1/2Q'$, the partitions can be merged locally for GO phase without any global shuffles across nodes, thereby minimizing the execution time. Afterwards, the reduced number of Q'' will increase the network communication overheads, and thus slow down the holistic computing. This is identical to our theoretical discussion in Section 6.3 and Appendix A. The trade-off between the scale of parallelism Q' and the communication overhead of Q''/Q' further motivates us to better configure our algorithm based on the hardware characteristics.

However, the corresponding fitness value for different Q' and Q''/Q', as in Fig. 8(b), indicates the enlarged partition number during the fitness calculation phase has an indirect impact on the quality of the optimization solution. In fact, the overall effectiveness is merely relevant to the quality of generated individuals, thus the fitness value is predominantly influenced by the proportion of elites in the GO phase. This will result in weak correlation between the utility value and the Q'. Assuming the total population size and Q' is fixed, the decreased Q'' indicates the growth of sub-population per partition in the subsequent GO phase. In this case, the proportion of elite individuals in the population descends. From Fig 8(b), we observe a slight decrease of fitness value at the beginning followed by an increase with the decrease of Q''. Therefore, GA-Par trades off the population diversity through controlling Q'' to find suitable parameters.

7.6 Scalability Evaluation

We evaluate how the machine number involved in GA-Par affects the execution performance. We fix the application type as *App(100, 100)* and the total population is initialized as 5000. Because of the relationship between Q' and Q'' found in Section 7.5, we only conduct the experiments under the pre-assumption of $Q'' = Q'/2$ with fixed $Q' = 160$.

Figure 9 demonstrates the scalability of our algorithm if the machine number is scaled from 5 to 20 nodes, where each node maintains the same number of partitions. In this manner, the total partition number will be proportional to the number of nodes. Due to the increased number of partitions, it leads to less computation workload involved within a partition. As a result, the increased level of parallelization can be achieved, substantially reducing the consumed time as shown in Fig. 9(a). Regarding the achievable optimal level, as demonstrated in Fig. 9(b), there is no direct correlation between the machine number and the fitness value. Despite the fact taht increased machines will result in the increment of partition number Q', the varying partition number Q'' will not directly impact on the fitness value (similar to the explanation for Fig. 8(b)).

Figure 10 describes another case where we fix the number of total partitions in different experiments. We can observe a consistent time reduction with the increment of the cluster size. Apparently, the increased computation capability from adding more machines will considerably accelerate the computation over a fixed population and partition number, in spite of increased communication overheads among partitions. Furthermore, for the aforementioned reason, the number of machines will not affect the fitness value.

8 Related Work

To deal with load spikes and vendor locks-in issue whilst fully exploiting the cloud diversities with reduced cost and guaranteed QoS, geo-distributed cloud techniques and architectures are proposed recently such as SuperCloud [37][38], MultiCloud[39], JointCloud [1]. In many cases, the use of sound security engineering techniques can reduce risks. An alternative approach is to directly utilize the reliable platform-as-a-service (PaaS) components provided by different Cloud providers, instead of designing and building applications from scratch. [11] [10] introduced a set of algorithms to orchestrate applications over federated Clouds. However, the security level of each microservice is randomly assigned. [12] assigned the security level by the performance of a specific security technique such as encryption algorithm applied in a Cloud-based service. However, it is extremely difficult to handle the Cloud diversities. Additionally, the complexity of large-scale geo-distributed Cloud environments results in a great number of uncertainties which cannot be modeled by the availability in conventional information security approaches [40][19]. In this paper, we propose a generic method to quantitatively measure the security level and its availability aspects under uncertainties of Cloud-based service components. Furthermore, how to optimize the service selection or component placement to meet different requirements has become a research hotspot in the last few years. [41] [42] [43] focused on QoS-aware service composition and its implementation in middleware. The QoS attributes typically include the response time, availability, reliability, price, and reputation, etc., without quantitatively depicting the secure objectives with risks and solving them in real Cloud environments. Meanwhile, the above reviewed works have not yet considered the uncertainties within federated Clouds and the corresponding impacts of massive-scale microservices on the security aspects. These motivate us to combine them in the emerging microservice orchestration to realize a reliable service-oriented system.

As for the optimization solution, solvers such as Mixed-Integer Non-linear Programming (MINLP) or Linear Programming (MIP) etc. are infeasible in the proposed large orchestration scenario due to the high time complexity [13]. Genetic Algorithm has been widely used to optimize resource management and task placement [44][45], and parallel Genetic Algorithms also have been explored for decades [14]. However, there is no general and easily-used framework that allows for implementing the GA over large-scale clusters. [46] first proposed how to implement GA over Hadoop and conducted detailed studies on the factors that would affect the algorithms' scalability. Similarly, this paper also studies the scalability factors of GA-Par within in-memory computing scenarios. The most recent works [32] [47] introduced a general framework using a map reduce paradigm, but neglected the linked data among interdependent or interactive Cloud services and their non-functional requirements. By contrast, GA-Par can be easily adapted into distributed models to accelerate the orchestration. [48] proposed an architecture to deploy and execute parallel GAs based on the available resources over Clouds. This architecture can be considered as an effective supplement to GA-Par.

9 Conclusion

In this paper, we describe a new microservice orchestration framework by considering both internal security threats and environmental uncertainties. The internal security satisfaction levels stem from application's topology and diverse provisioning capability of microservice candidates. The uncertainties are captured by black-boxed modelling in terms of response time and transmission rates over Clouds. We adopt a novel paralleled GA-Par on Spark to accelerate the solving of the proposed orchestration. Some important findings and conclusions can be drawn as follows:

- *Improving dependability of massive scale orchestration is becoming of increasing importance*. Quantitatively formalized modeling and measurement throughout both internal and external factors can comprehensively empower maximized achievability of dependability.
- *Standing on the security viewpoint reveals an effective means to measure the system dependability.* Although we focus on the security-aware modeling and problem formulation, the presented methodology can be easily applied into other dimensions for depicting dependability such as reliability, availability and safety, etc.
- *Relying on real dataset and data-driven approach is critical to understanding the real-world problems and formulating assumptions under realistic circumstances.* Statistic approaches can be exploited to deal with the long-standing environmental uncertainty issues and it is also an effective means to capture the long-tail response characteristics of Cloud-based services.
- *Integrating GA-Par with IoT and Fog eco-system to facilitate the service orchestration.* Within the coming decades, the concept of the exascale system will become increasingly commonplace, interconnecting billions of different sensors, things and other data sources across a vast number of industries which will likely co-exist in some form of Fog and smart mobility eco-system [49][50]. The challenges of orchestration pertaining to security and uncertainties will continue to play a critical concern for designing these systems.

More practical validations are underway to demonstrate the industrialized process into production-level cluster scheduling and container orchestration systems based on collaborative works with Alibaba such as [51][52]. Technically, we will further consider data partition to achieve individual reduction of GA incurred by partitioning the data to different Slaves. Finally, studies are also needed to develop partition algorithms to make GA-Par more efficient.

Appendix A
Performance analysis

A.1 Spark-based Execution Model

Apache Spark is built upon the concept of Resilient Distributed Datasets (RDD) and provides actions/transformations on top of RDD. More precisely, Spark applications include three steps: creating new RDDs, transforming existing RDDs, and calling operations/actions on RDDs to compute the results.

A Spark application consists of a *driver* process and a set of *executor* processes scattered across nodes on a cluster. A *driver* is a process that controls the high-level data flow, while *executor* processes are responsible for executing tasks. Batches of tasks run concurrently on each *executor* throughout the application's lifetime. In other words, these tasks are distributed to RDD partitions in a bijective way and are executed concurrently. As a result, the parallelism of Spark is highly dependent upon the partition number as well as the *executor*'s capability. Using Spark command line flags is a flexible approach for tuning the utilization of computational resources to available CPU cores. For example, the "executor-core" flag can specify the number of CPU cores bound to each *executor*, controls which the number of concurrent tasks. Suppose a cluster has N slaves and each has C CPU cores. If we parallelize the execution of NC tasks over the cluster, the average utility is $\frac{NC}{T}$.

The performance of GA-Par is evaluated with respect to the following three factors: *computation complexity, network communication, and disk reads/writes*. For ease of exposition, we simplify the time complexity analysis by first considering one iteration of the algorithm.

A.2 Computation Complexity

The computation is the main constituent of the overall time complexity. For sequential GA computation, the time complexity of each iteration can be divided into the following:

Population Creation/Reproduction. According to our definition in Spark preliminary, Q' partitions correspond to Q' tasks to be executed. Each partition involves $\frac{M}{Q'}\sum_{i=1}^{L}|S_i|$ computations. Therefore we have:

$$T1 = Q' \times (\frac{M}{Q'} \times \sum_{i=1}^{L} |S_i|) \approxeq \mathcal{O}(M|S|)$$

Fitness Score Computation. The computation of each partition can be summarized as $\frac{M}{Q'} \times t_1$, where t_1 is the time complexity of workflow evaluation. More details are as follows:

$$\begin{aligned} t_1 &= \sum_{s_{jh}\in\lambda'} Value(c_i, s_{jh}, mrt, t) \\ &= \sum_{s_{jh}\in\lambda'} T(mrt, s_{jh}) \cdot \{\theta \cdot \varepsilon(t_{s_{jh}}) \cdot \frac{Data_{S_j}}{Total} \\ &+ (1-\theta) \cdot [VDDSD(c_i, s_{jh}) + DSSD(c_i, s_{jh})]\} \end{aligned}$$

Thus, the time complexity t_1 can be written as

$$\begin{aligned} t_1 = \sum_{s_{jh}\in\lambda'} T(mrt, s_{jh}) \cdot \{\theta \cdot [\varepsilon(t_{s_{jh}}) \cdot \frac{Data_{S_j}}{Total}]+ \\ (1-\theta) \cdot [\sum_{\{d_{s_{ip}\to s_{jh}}\}\in IN_{s_j}} \sum_{q=1}^{|G|} w_{ij}^q \cdot M(g_{s_{ip}}^q, g_{s_{jh}}^q)+ \\ \sum_{q=1}^{|G|} w_j^q \cdot M(g_{c_i}^q, g_{s_{jh}}^q)]\} \end{aligned} \quad (19)$$

and approximated as follows:

$$\begin{aligned} t_1 &\approxeq \sum_{i=1}^{L}(C_1 + \sum_{j=1}^{|IN_{S_j}|} C_2 + C_2) \approxeq C_1L + C_2|IN|L + C_2L \\ &\approxeq (C_1 + C_2)L + C_2|IN|L \end{aligned}$$

where C_1 and C_2 are constants. As a result, IN_{S_j} represents the input data from S_j's immediate predecessors, and IN is the set of all data dependencies. In the best case, we have $|IN| = L - 1$, while in the worst case the number of data dependencies is $|IN| = L(L-1)$. Consequently, for Q' partitions, the time complexity $T2$ is:

$$T2 \approxeq Q' \times (\frac{M}{Q'} \times t_1) \approxeq \mathcal{O}(ML^3)$$

Updating the Elitism List. The time complexity of finding the elitism list on a single host is $\mathcal{O}(Mlog(M))$. However, the time complexity can be reduced by first generating the local elitism list for each partition. A local elitism list is obtained by running the top E nearest neighbors on each partition. The global elitism list with size E can then be generated by sorting EQ' individuals.

Since each partition has $\frac{M}{Q'}$ individuals, the time used to sort the fitness value in each partition can be approximated to $\mathcal{O}(\frac{M}{Q'}log(\frac{M}{Q'}))$, and finding the global elitism list requires $\mathcal{O}(EQ'log(EQ'))$. Thus, the time complexity of updating the global elitism list is:

$$\begin{aligned} T3 &\approxeq \mathcal{O}(Q' \times (\frac{M}{Q'}log(\frac{M}{Q'})) + EQ'log(EQ')) \\ &\approxeq \mathcal{O}(Mlog(\frac{M}{Q'}) + EQ'log(EQ')) \end{aligned}$$

Crossover. Since *Genetic Operations* (GO) works on a smaller number of partitions Q'' to ensure the diversity of the next generation, the time complexity of both Crossover and Mutation should be considered under this case. The population size of each partition is $\frac{M}{Q''}$, which reduces the time complexity of Crossover to:

$$T4 \approxeq Q'' \times (\frac{M}{2Q''} \cdot L)s \approxeq \mathcal{O}(ML)$$

Mutation. Since it involves the rate of Mutation, the time complexity can be approximated as:

$$T5 \approxeq \mathcal{O}(M)$$

In summary, the time complexity for each iteration is:

$$\begin{aligned} &Time(Computation) = T_1 + T_2 + T_3 + T_4 + T_5 \\ &\approxeq \mathcal{O}(M|S|) + \mathcal{O}(ML^3)) + \mathcal{O}(Mlog(\frac{M}{Q'}) + EQ'log(EQ')) \\ &\quad + \mathcal{O}(ML) + \mathcal{O}(M) \\ &\approxeq \mathcal{O}(M(|S| + L^3 + log(\frac{M}{Q'})) + EQ'log(EQ')) \end{aligned}$$

Based on the above analysis, we prove that the overall GA-Par computation can be reduced to $\mathcal{O}(\frac{T}{NC} \times Time(Computation))$, which clearly demonstrates the scalability of the algorithm.

Proof. Assume we have a N-slaves Cluster, each slave has C cores, and each task needs T cores of CPU. Then, there are $\frac{NC}{T}$ execution slots that can run in parallel, i.e. the speed-up is $\frac{T}{NC}$. Since $Time(Computation)$ represents the total workload of a single machine with one single-core CPU, the time complexity of our algorithm can reduce to $\mathcal{O}(\frac{T}{NC} \times Time(Computation))$. □

A.3 Network Communication

Shuffling also affects execution time, so data transmission and partition operations should be carefully considered. We simplify our evaluation model by assuming that our ideal Spark implementation will not block the Spark program and only the observable and countable network communication delay is considered:

Shuffling Cost of Updating the Elitism List. This procedure includes collection of the local elitism lists and broadcast of the global elitism list. During collection of the local elitism lists, we primarily consider the maximal network delay $\max_{i\in[1,Q']}(dCollect_i)$, where $dCollect_i$ is the time delay for collecting the local elitism list from the i^{th} partition. Similarly, for broadcasting, the time delay should be $\max_{j\in[1,N]}(dBroadcast_j)$, where $dBroadcast_j$ is the time delay for broadcasting the global elitism list to the j^{th} slave. As a reminder, either the size of local elitism list or global elitism list is E. Thus, we can evaluate the cost as T_6 where C_4, C_5 and C_6 are constants:

$$\begin{aligned} T_6 &= C_4 \max_{i\in[1,Q']}(dCollect_i) + C_5 \max_{j\in[1,N]}(dBroadcast_j) + C_6 \\ &\approxeq \mathcal{O}(E) \end{aligned}$$

Shuffling Cost of Partition Adjustment. The performance depends on the value of Q' and Q'', where

$$Q' = \lfloor\frac{Q'}{N}\rfloor \times a_1 + b_1; Q'' = \lfloor\frac{Q''}{N}\rfloor \times a_2 + b_2$$

Based on the scheme mentioned above, at most $\lfloor\frac{Q'}{Q''}\rfloor$ partitions can be merged into a single partition. Also, this merging process is not limited to the same slave and might occur among the partitions in different slaves. This will cause the data shuffling. In particular:

- If $\lfloor \frac{Q'}{N} \rfloor = i \times \lfloor \frac{Q'}{Q''} \rfloor$ ($i \in \mathbf{Z}$), no additional shuffling occurs because the merging process operates within the same slave.
- Otherwise, the partitions will cross over slaves. Assume $\lfloor \frac{Q'}{N} \rfloor = i \times \lfloor \frac{Q'}{Q''} \rfloor + r$, where $i, r \in \mathbf{Z}$. The worst case scenario of partition adjustment would be Q' partitions each containing $\frac{M}{Q'}$ individuals, participating in the shuffling procedure. The communication cost is $C_7 \cdot Q' \cdot \frac{M}{Q'} \times \theta$, i.e., $C_7 M \theta$ where θ is the bit transmission rate for an individual. The worst case scenario rarely happens under the control of the optimized Spark shuffling schema, but in the best case, each Slave only involves the remaining r partitions. The ideal shuffling cost would be $C_8 N \times (\lfloor \frac{Q'}{N} \rfloor \mod \lfloor \frac{Q'}{Q''} \rfloor)$.

In summary, the shuffling cost T_7 for partition adjustment would fall within the following:

$$C_8 N \times (\lfloor \frac{Q'}{N} \rfloor \mod \lfloor \frac{Q'}{Q''} \rfloor) \leq T_7 \leq C_7 M \theta$$

A.4 Disk Reads/Writes

Our system involves intensive computations but the processed data can easily fit into memory. As a result, we can ignore the cost of I/O in the initialization phase. The shuffling process will involve some unavoidable I/O costs, which can follow the partial schema of A.3. Specifically, the analysis includes:

Updating the Elitism List. We first collect the local elitism lists from each partition and read them on the Master node. the I/O cost is $2EQ'$. Similarly, the I/O cost for broadcasting the global elitism list is $1 + Q''$. As a result, the total I/O cost in this phase is $2EQ' + Q'' + 1$.

Partition Adjustment. Based on the analysis in A.3, the I/O cost is approximately between $2N(\lfloor \frac{Q'}{N} \rfloor \mod \lfloor \frac{Q'}{Q''} \rfloor)$ and $2M$.

Acknowledgment

This work is supported by the National Key Research and Development Program (2016YFB1000103) and the National Natural Science Foundation of China (61421003). This work is also supported by Beijing Advanced Innovation Center for Big Data and Brain Computing (BDBC). Renyu Yang is the corresponding author.

References

[1] H. Wang, P. Shi, and Y. Zhang, "Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization," in *IEEE ICDCS*, 2017.

[2] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos, "Fog orchestration for internet of things services," *IEEE Internet Computing*, vol. 21, no. 2, pp. 16–24, 2017.

[3] A. C. Zhou, Y. Gong, B. He, and J. Zhai, "Efficient process mapping in geo-distributed cloud data centers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 16.

[4] A. C. Zhou, S. Ibrahim, and B. He, "On achieving efficient data transfer for graph processing in geo-distributed datacenters," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 1397–1407.

[5] M. Natu, R. K. Ghosh, R. K. Shyamsundar, and R. Ranjan, "Holistic performance monitoring of hybrid clouds: Complexities and future directions," *IEEE Cloud Computing*, vol. 3, no. 1, pp. 72–81, 2016.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE TDSC*, vol. 1, no. 1, pp. 11–33, 2004.

[7] A. Dusia, Y. Yang, and M. Taufer, "Network quality of service in docker containers," in *2015 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2015, pp. 527–528.

[8] Y. Chen, A. Gorbenko, V. Kharchenko, and A. Romanovsky, *Measuring and Dealing with the Uncertainty of SOA Solutions*. IGI Global, 2012.

[9] A. Gorbenko and A. Romanovsky, "Time-outing internet services," *IEEE Security & Privacy*, 2013.

[10] P. Watson, "A multi-level security model for partitioning workflows over federated clouds," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 1, no. 1, p. 15, 2012.

[11] Z. Wen, J. Cala, P. Watson, and A. Romanovsky, "Cost effective, reliable and secure workflow deployment over federated clouds," *IEEE TSC*, 2016.

[12] W. Liu, S. Peng, W. Du, W. Wang, and G. S. Zeng, "Security-aware intermediate data placement strategy in scientific cloud workflows," *Knowl. Inf. Syst.*, 2014.

[13] S. Malek, N. Medvidovic, and M. Mikic-Rakic, "An extensible framework for improving a distributed software system's deployment architecture," *IEEE TSE 2012*.

[14] E. Cantú-Paz, "A summary of research on parallel genetic algorithms," 1995.

[15] Y. Gan, Y. Zhang, D. Cheng *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in *ACM ASPLOS 2019*.

[16] N. R. Potlapally, S. Ravi, A. Raghunathan, and N. K. Jha, "A study of the energy consumption characteristics of cryptographic algorithms and security protocols," *IEEE Transactions on mobile computing*, vol. 5, no. 2, pp. 128–143, 2006.

[17] S. Ji, W. Li, P. Mittal, X. Hu, and R. Beyah, "Secgraph: A uniform and open-source evaluation system for graph data anonymization and de-anonymization," in *USENIX Security*, 2015.

[18] D. Catteddu and G. Hogben, "Cloud computing: Benefits, risks and recommendations for information security," enisa:European Network and Information Security Agency, Tech. Rep., 2009.

[19] M. E. Whitman and H. J. Mattord, *Principles of Information Security*, 2004.

[20] D. Trihinas, A. Tryfonos, M. D. Dikaiakos, and G. Pallis, "Devops as a service: Pushing the boundaries of microservice adoption," *IEEE Internet Computing*, vol. 22, no. 3, pp. 65–71, 2018.

[21] T. Xie and X. Qin, "Performance evaluation of a new scheduling algorithm for distributed systems with security heterogeneity," *Journal of Parallel and Distributed Computing*, 2007.

[22] M. Forshaw, "Operating policies for energy efficient large scale computing," Ph.D. dissertation, Newcastle University, UK, 2015.

[23] M. Dobber, R. van der Mei, and G. Koole, "Effective prediction of job processing times in a large-scale grid environment," in *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*. IEEE, 2006, pp. 359–360.

[24] T. Miu and P. Missier, "Predicting the execution time of workflow activities based on their input features," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 64–72.

[25] R. J. Adler, R. E. Feldman, and M. S. Taqqu, Eds., *A Practical Guide to Heavy Tails: Statistical Techniques and Applications*, 1998.

[26] A. B. Downey, "Evidence for long-tailed distributions in the internet," in *SIGCOMM*, 2001.

[27] V. P. Chistyakov, "A theorem on sums of independent positive random variables and its applications to branching random processes," *Theory of Probability & Its Applications*, 1964.

[28] E. Seneta, Ed., *Regularly Varying Functions*. Springer, 1976.

[29] P. Embrechts, T. Mikosch, and C. Klüppelberg, *Modelling Extremal Events: For Insurance and Finance*, 1997.

[30] P. Cirillo, "Are your data really pareto distributed?" *Physica A: Statistical Mechanics and its Applications*.

[31] D. Bhandari, C. Murthy, and S. K. Pal, "Genetic algorithm with elitist model and its convergence," *IJPRAI*, 1996.

[32] P. Salza, F. Ferrucci, and F. Sarro, "elephant56: Design and implementation of a parallel genetic algorithms framework on hadoop

mapreduce," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. ACM, 2016, pp. 1315–1322.

[33] M. Zaharia, M. Chowdhury, T. Das, and et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.

[34] C. Murthy, D. Bhandari, and S. K. Pal, "ε-optimal stopping time for genetic algorithms," *Fundamenta Informaticae*, vol. 35, no. 1-4, pp. 91–111, 1998.

[35] Y. Zhang, Z. Zheng, and M. R. Lyu, "Wspred: A time-aware personalized qos prediction framework for web services," in *ISSRE*, 2011.

[36] "Dataset used for the experiments," https://github.com/lukewen427/GA-Par-data, accessed: 2019-04-25.

[37] Z. Shen, Q. Jia, G.-E. Sela, W. Song, H. Weatherspoon, and R. Van Renesse, "Supercloud: A library cloud for exploiting cloud diversity," *ACM TOCS*, 2017.

[38] Q. Jia, Z. Shen, W. Song, R. Van Renesse, and H. Weatherspoon, "Supercloud: Opportunities and challenges," *ACM SIGOPS Operating Systems Review*, 2015.

[39] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, and L. Seinturier, "A federated multi-cloud paas infrastructure," in *IEEE Cloud*. IEEE, 2012.

[40] D. B. Parker, *Fighting Computer Crime: A New Framework for Protecting Information*, 1998.

[41] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient algorithms for web services selection with end-to-end qos constraints," *ACM Trans. Web*, 2007.

[42] M. Alrifai, D. Skoutas, and T. Risse, "Selecting skyline services for qos-based web service composition," in *Proceedings ACM WWW*. ACM, 2010, pp. 11–20.

[43] S. Rosario, A. Benveniste, S. Haar, and C. Jard, "Probabilistic qos and soft contracts for transaction-based web services orchestrations," *IEEE TSC*, 2008.

[44] C. Guerrero, I. Lera, B. Bermejo, and C. Juiz, "Multi-objective optimization for virtual machine allocation and replica placement in virtualized hadoop," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 11, pp. 2568–2581, 2018.

[45] Z. Zhu, G. Zhang, M. Li, and X. Liu, "Evolutionary multi-objective workflow scheduling in cloud," *IEEE Transactions on parallel and distributed Systems*, vol. 27, no. 5, pp. 1344–1357, 2016.

[46] A. Verma, X. Llorà, D. E. Goldberg, and R. H. Campbell, "Scaling genetic algorithms using mapreduce," in *Intelligent Systems Design and Applications, 2009. ISDA'09. Ninth International Conference On*. IEEE, 2009, pp. 13–18.

[47] F. Ferrucci, P. Salza, and F. Sarro, "Using hadoop mapreduce for parallel genetic algorithms: A comparison of the global, grid and island models," *Evolutionary computation*, no. Early Access, pp. 1–33, 2017.

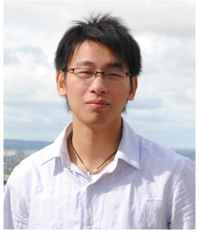
[48] P. Salza, F. Ferrucci, and F. Sarro, "Develop, deploy and execute parallel genetic algorithms in the cloud," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. ACM, 2016, pp. 121–122.

[49] A. Longo, M. Zappatore, M. Bochicchio, and S. B. Navathe, "Crowd-sourced data collection for urban monitoring via mobile sensors," *ACM Transactions on Internet Technology (TOIT)*, 2017.

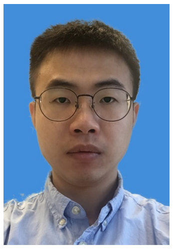
[50] A. Longo, M. Zappatore, and S. B. Navathe, "The unified chart of mobility services: Towards a systemic approach to analyze service quality in smart mobility ecosystem," *Journal of Parallel and Distributed Computing*, 2019.

[51] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: a fault-tolerant resource management and job scheduling system at internet scale," *Proceedings of the VLDB Endowment*, 2014.

[52] X. Sun, C. Hu, R. Yang, P. Garraghan, T. Wo, J. Xu, J. Zhu, and C. Li, "Rose: Cluster resource scheduling via speculative over-subscription," in *IEEE ICDCS*, 2018.



Zhenyu Wen is currently a postdoc researcher with the School of Computing, Newcastle University, UK. He received M.S and Ph.D. degrees in computer science from Newcastle University, Newcastle Upon Tyne, UK in 2011 and 2015 respectively. His current research interests include Multi-objects optimisation, Crowdsources, AI and Cloud computing.

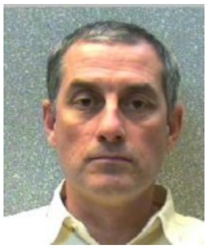


Tao Lin is currently a Ph.D. student at EPFL, Switzerland. Prior to that, he received his M.sc and B.E. from EPFL and Zhejiang University. His research interests include distributed machine learning, optimization etc.

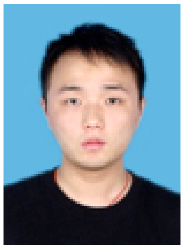
Renyu Yang is a Research Fellow with University of Leeds, UK. He is also an adjunct research scientist with BDBC, Beihang University, China. He received his Ph.D. degree from Beihang University. He was previously with Alibaba Group and had industrial experience in building large-scale systems. His research interests include reliable resource management, distributed systems, data processing, etc. He is a member of IEEE.

Shouling Ji is a Professor at Zhejiang University and a Research Faculty in the School of Electrical and Computer Engineering at Georgia Institute of Technology (Georgia Tech). He received two Ph.D. degrees from Georgia Institute of Technology and Georgia State University, and B.S. (with Honors) and M.S. degrees from Heilongjiang University. His current research interests include data-driven security and privacy, AI security and big data analytics. He is a member of IEEE.

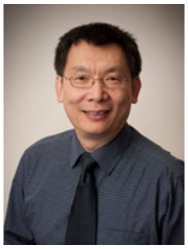
Rajiv Ranjan is a Professor in Computing Science at Newcastle University, UK. Before moving to Newcastle University, he was Julius Fellow (2013-2015), Senior Research Scientist and Project Leader in the Digital Productivity and Services Flagship of Commonwealth Scientific and Industrial Research Organization. Prior to that he was a Senior Research Associate (Lecturer level B) in the School of Computer Science and Engineering, University of New South Wales. He has a Ph.D. from the department of Computer Science and Software Engineering, the University of Melbourne.



Alexander Romanovsky Alexander Romanovsky is a Professor in Computing Science at Newcastle University, UK. He is the investigator of the EPSRC platform grant on Layers for Structuring Trustworthy Ambient Systems (STARTA) and a co-investigator of the EPSRC PRiME programme grant on Power-efficient, Reliable, Many-core Embedded systems. Before this, he coordinated the major EU FP7 DEPLOY IP that developed the Rodin tooling environment for formal stepwise design of complex dependable systems using Event-B. His main research areas are system dependability, fault tolerance, safety, modelling and verification.



Changting Lin is currently a Ph.D. candidate at Zhejiang University. He received B.E. and M.S. degrees from Zhejiang Gongshang University in 2009 and 2012. His research interests include Software Defined Network, reconfigurable and network security.



Jie Xu is Chair Professor of Computing at University of Leeds, Director of UK EPSRC WRG e-Science Centre, Chief Scientist of BDBC, Beihang University, China. He has industrial experience in building large-scale networked systems and has worked in the field of dependable distributed computing for over 30 years. He is a Steering/Executive Committee member for numerous IEEE conferences including SRDS, ISORC, HASE, SOSE and is a co-founder for IEEE IC2E. He has led or co-led many research projects to the value of over $30M, and published in excess of 300 academic papers, book chapters and edited books. He is a member of IEEE.