Minimizing Redundancy to Satisfy Reliability Requirement for a Parallel Application on Heterogeneous Service-Oriented Systems

Guoqi Xie, *Member, IEEE*, Gang Zeng, *Member, IEEE*, Yuekun Chen, Yang Bai, Zhili Zhou, Renfa Li, *Senior Member, IEEE*, and Keqin Li, *Fellow, IEEE*

Abstract—Reliability is widely identified as an increasingly relevant issue in heterogeneous service-oriented systems because processor failure affects the quality of service to users. Replication-based fault-tolerance is a common approach to satisfy application's reliability requirement. This study solves the problem of minimizing redundancy to satisfy reliability requirement for a directed acyclic graph (DAG)-based parallel application on heterogeneous service-oriented systems. We first propose the enough replication for redundancy minimization (ERRM) algorithm to satisfy application's reliability requirement, and then propose heuristic replication for redundancy minimization (HRRM) to satisfy application's reliability requirement with low time complexity. Experimental results on real and randomly generated parallel applications at different scales, parallelism, and heterogeneity verify that ERRM can generate least redundancy followed by HRRM, and the state-of-the-art MaxRe and RR algorithm. In addition, HRRM implements approximate minimum redundancy with a short computation time.

Index Terms—Fault-tolerance, heterogeneous service-oriented systems, quality of service, reliability requirement, replication

1 Introduction

1.1 Background

Cloud-based service is a new service-based resource sharing paradigm [1], [2]. In X as a service (XaaS) clouds, resources as services (e.g., infrastructure, platform and software as a service) are sold to applications such as scientific and big data analysis workflows [1], [3], [4], [5], [6]. Meanwhile, cloud computing systems become more heterogeneous as old, slow machines are continuously replaced with new, fast ones. Heterogeneous computing systems consist of diverse sets of processors interconnected with a high-speed network, and are applied in business-critical, mission-critical, and safety-critical scenarios to achieve operational goals [7]. Applications in the system are increasingly parallel and the tasks in an application have obvious data dependencies and precedence constraints [1], [8], [9], [10], [11]. Examples of parallel applications are Gaussian elimination and fast Fourier transform [9]. A parallel application with precedence constrained tasks at a high level is described by a directed acyclic graph (DAG) [1], [8], [9], [10], [11], where nodes represent tasks, and edges represent communication messages between tasks. Such application is usually called DAG-based parallel application [12].

The current cloud-based service systems are actually heterogeneous service-oriented systems where resource management is a considerable challenge owing to the various configurations or capacities of the hardware or software [13]. The processing capacity of processors in heterogeneous service-oriented systems has been developed to provide powerful cloud-based services, whereas failures of processors will affect the reliability of systems and quality of service (QoS) for users [2]. Reliability is defined as the probability of a schedule successfully completing its execution, and it has been widely identified as an increasingly relevant issue in service-oriented computing systems [2], [14], [15], [16], [17], [18].

Fault-tolerance by primary-backup replication, which means that a primary task will have zero, one, or multiple backup tasks, is an important reliability enhancement mechanism. In the primary-backup replication scheme, the primary and all the backups are called replicas. Although replication-based fault-tolerance is an important reliability enhancement mechanism [14], [15], [19], [20], [21], any application cannot be 100 percent reliable in practice. Therefore, if an application can satisfy its specified reliability requirement (also named reliability goal or reliability assurance in some studies), then it is considered to be reliable. For example, assume that the application's reliability requirement is 0.9, only if the application's reliability exceeds 0.9, will the application be reliable. Specifically,

- G. Xie, Y. Chen, Y. Bai, and R. Li are with the College of Computer Science and Electronic Engineering, Hunan University, Key Laboratory for Embedded and Network Computing of Hunan Province, Hunan 410082, China. E-mail: {xgqman, baiyang, lirenfa}@hnu.edu.cn, chenyuekun@126.com.
- G. Zeng is with the Graduate School of Engineering, Nagoya University, Aichi 4648603, Japan. E-mail: sogo@ertl.jp.
- Z. Zhou is with the Nanjing University of Information Science and Technology, Nanjing 210044, China. E-mail: zhou_zhili@163.com.
- K. Li is with the College of Computer Science and Electronic Engineering, Hunan University, Key Laboratory for Embedded and Network Computing of Hunan Province, Hunan 410082, China, and the Department of Computer Science, State University of New York, New Paltz, NY 12561. E-mail: lik@newpaltz.edu.

Manuscript received 14 Aug. 2016; revised 1 Jan. 2017; accepted 3 Feb. 2017. Date of publication 0 . 0000; date of current version 0 . 0000.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSC.2017.2665552

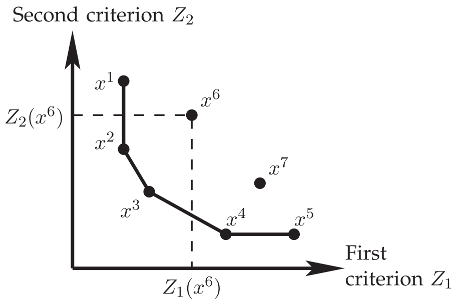


Fig. 1. Pareto optima and pareto curve for a bicriteria minimization problem [19].

reliability requirement has been defined in some reliability related standards (e.g., IEC 61508 [22] and ISO 9000 [23]), and it is one of the most important QoS in cloud and services computing systems [14], [15]. Therefore, reliability requirement must be satisfied from standards and QoS perspectives. However, as pointed out in [2], many cloud-based services failed to fulfill their reliability requirements due to processor failures in practice.

1.2 Motivation

Users and resource providers are the two types of roles with different requirements for service-oriented systems [24]. For users, satisfying application's reliability requirement is one of the most important QoS requirements, for which replication-based fault-tolerance is a common approach. For resource providers, minimizing resource redundancy caused by replication is one of the most important concerns [14], [15]. However, adding more replicas (including primary and backups) could increase both reliability and redundancy for a parallel application. Therefore, both criteria (low redundancy and high reliability, short schedule length and high reliability) are conflicting, and optimizing them is a bi-criteria optima problem [19]. In Fig. 1, each point x^1-x^7 represents a solution of a bicriteria minimization problem [19]. The points x^1, x^2, x^3, x^4, and x^5 are Pareto optima [25]; the points x^1 and x^5 are weak optima, whereas the points x^2, x^3, and x^4 are strong optima. The set of all Pareto optima is called the Pareto curve [19]. Many studies have dealt specifically with the bi-criteria (i.e., minimizing schedule length and maximizing reliability) problem to obtain such an approximate Pareto curve for a DAG-based parallel application [19], [20], [21], [26], [27], [28]. In [26], [27], [28], the approaches increase reliability by efficient task scheduling without using replication. In [19], [20], [21], the approaches presented replicate tasks to increase reliability.

However, for heterogeneous service-oriented systems, resolving the above bi-criteria is not strictly required for the following reasons:

(1) Clouds allow flexible and dynamic resource allocations based on a pay-as-you-go scheme [29], where users pay only for the reliability requirement they apply and will not pay additional fees for the reliability that surpasses their reliability requirement.
(2) The application cannot be 100 percent reliable as mentioned earlier. The most common component of service-level agreement (SLA) between resource providers and the users is that the services (reliability requirement in this study) should be provided to the users as agreed upon in the contract [30]. Therefore, satisfying application's reliability requirement is the service level objective.

In summary, considering the actual demand, the theoretical bi-objective optimization problem could be degradated to a constrained single-objective optimization problem in most cases. In other words, reliability is not the higher the better, but as long as you can satisfy the reliability requirement from a practical perspective. Therefore, the reliability problem of service-oriented systems is mainly to satisfy application's reliability requirement while still reducing the resource as far as possible.

The approaches related to our work are [14] and [15], in which the authors presented the MaxRe and RR algorithms to minimize redundancy of a parallel application to satisfy application's reliability requirement on heterogeneous distributed systems. The main procedures of the MaxRe and RR are follows:

(1) The reliability requirement of the application is transferred to the sub-reliability requirements of the tasks. In this way, as long as the sub-reliability requirement of each task can be satisfied, the application's reliability requirement can be satisfied, such that a heuristic replication can be used in the following.
(2) MaxRe and RR iteratively assign the replicas of each task to the processors with maximum reliability values until the sub-reliability requirement of the task is satisfied.

However, the essential limitation of MaxRe and RR is that the sub-reliability requirements of tasks are too high, thereby causing them need unnecessary redundancy to satisfy the sub-reliability requirements.

1.3 Our Contributions

Similar to the state-of-the-art MaxRe and RR, this study aims to implement redundancy minimization to satisfy application's reliability requirement for a parallel application on heterogeneous service-oriented distributed systems. Our contributions comparing to the MaxRe and RR are summarized as follows:

(1) We present the just enough replication for redundancy minimization (ERRM) algorithm to satisfy application's reliability requirement by two-stage replications. The first stage involves obtaining the lower bound on redundancy (i.e., the minimum required number of replicas) for each task; the second stage is iteratively selecting the available replicas and corresponding processors with the maximum reliability values until application's reliability requirement is satisfied.
(2) To overcome the high time complexity of ERRM algorithm, we propose the heuristic replication for redundancy minimization (HRRM) algorithm to deal with large-scale parallel applications. Similar to the MaxRe and RR algorithms, HRRM first transfers the reliability requirement of the application to the sub-reliability requirements of the tasks. Then, HRRM iteratively assign the replicas of each task to the processors with maximum reliability values until

the sub-reliability requirement of the task is satisfied. The main improvement of HRRM over MaxRe and RR is that it can obtain lower sub-reliability requirements for most tasks, such that HRRM generates less redundancy than MaxRe and RR.

(3) Experimental results on real and randomly generated parallel applications at different scales, parallelism degrees, and heterogeneity degrees validate that ERRM can generate the least redundancy followed by HRRM, the state-of-the-art MaxRe and RR algorithm. In addition, HRRM implements approximate minimum redundancy with a short computation time.

The rest of this paper is organized as follows. Section 2 reviews related research. Section 3 presents the reliability modeling and problem statement. Section 4 explains the state-of-the-art MaxRe and RR algorithms. Sections 5 and 6 proposed the ERRM and HRRM algorithms, respectively. Section 7 verifies the ERRM and HRRM algorithms. Section 8 concludes this study.

2 Related Work

The widely-accepted reliability model was presented by Shatz and Wang [31], where each hardware component (processor) is characterized by a constant failure rate per time unit λ and the reliability during the interval of time t is $e^{-\lambda t}$. That is, the failure occurrence follows a constant parameter Poisson law [31]. This law is also known as the exponential distribution model [19]. This section mainly reviews the related research on reliability and fault-tolerance of DAG-based parallel applications.

Two main types of primary-backup replication approaches exist in current: active replication [14], [15], [20], [21] and passive replication [32], [33], [34], [35]. For the active replication scheme, each task is simultaneously replicated on several processors, and the task will succeed if at least one of them does not fail. For the passive scheme, whenever a processor fails, the task will be rescheduled to proceed on a backup processor. When a processor crashes, it is subsequently restarted to continue from the checkpoint just as if no failure had occurred; such scheme is called checkpoint and restart scheme, and can be considered as an improved version of the passive scheme [14], [15]. Meanwhile, according to the number of the backups, three types of primary-backup replication approaches exist; single backup for each primary, fixed ε backups for each primary, and quantitative backups for each primary.

The single backup for each primary approach is a simple method. Main representative methods include efficient fault-tolerant reliability cost driven (eFRCD) [33], efficient fault-tolerant reliability driven (eFRD) [34], and minimum completion time with less replication cost (MCT-LRC) [35] et al. Regarding their limitations, first, these approaches assume that no more than one failure happens at one moment; they are too ideal to tolerate potential multiple failures. Second, although passive replication also supports multiple backups for each primary [32], it is unsuitable for service-oriented applications; the reason is that once a processor failure is detected, the scheduler should reschedule the task located on the failed processor, and reassign it to a new processor, such that the QoS for the application is uncertain.

The fixed ε backups for each primary approach is an active replication approach, and is suitable for service-oriented systems because it can directly shield the failed tasks in performing, and the failure recovery time is almost close to zero [19], [20], [21]. In [19], the authors presented bicriteria scheduling heuristic (BSH) to minimize the schedule length of the application while taking the failure rate as a constraint; BSH can generate a Pareto curve of non-dominated solutions, among which the user can choose the compromise that fits his requirements best. However, the time complexity of BSH is as high as $O(n \times 2^u)$, where n is the number of replicas and u is the number of processors. In [20], Benoit et al. presented the fault-tolerant scheduling algorithm (FTSA) for a parallel application on heterogeneous systems to minimize the schedule length given a fixed number of failures supported in the system based on the active replication scheme. In [21], Benoit et al. further designed a new scheduling algorithm to minimize schedule length under both throughput and reliability constraints for a parallel application on heterogeneous systems based on the active replication scheme. The main problem in [20], [21] is that they need ε backups for each task with high redundancy to satisfy application's reliability requirement. Although application's reliability requirement can be satisfied by using active replication scheme, high redundancy causes high resource cost to resource providers.

Considering that fixed ε backups for each primary approach has high redundancy, recent studies begun to explore quantitative backups for each task approach to satisfy application's reliability requirement [14], [15]. Quantitative backups means different primaries have different numbers of backups, and the quantitative approach has lower resource cost than the fixed ε backups for each task based on active replication [14]. In [14] and [15], the authors proposed fault-tolerant scheduling algorithms MaxRe and RR; both MaxRe and RR incorporate reliability analysis into the active replication and exploit a dynamic number of backups for different tasks by considering each task's sub-reliability requirement. As discussed in Section 1.2, both MaxRe and RR have limitations in calculating the sub-reliability requirements of tasks. In [15], the authors also presented the DRR algorithm that extends RR by further considering the deadline requirement of a parallel application; however, we are only interested in satisfying reliability requirement in this study.

3 Reliability Modeling and Problem Statement

Table 1 gives the important notations and their definitions as used in this study.

3.1 Application Model

Let $U = \{u_1, u_2, \ldots, u_{|U|}\}$ represent a set of heterogeneous processors, where $|U|$ is the size of set U. In this study, for any set X, $|X|$ is used to denote size. A development life cycle of a service-oriented system usually involves the analysis, design, implementation, and testing phases. In this study, we focus on the design phase. Therefore, we assume that the processor and application parameter values are known in the design phase, because these values have been already calculated in the analysis phase.

TABLE 1
Important Notations in this Study

Notation	Definition
$c_{i,j}$	Communication time between the tasks n_i and n_j
$w_{i,k}$	Execution time of the task n_i on the processor u_k
$\overline{w_i}$	Average execution time of the task n_i
$rank_{\mathrm{u}}(n_i)$	Upward rank value of the task n_i
$\vert X\vert$	Size of the set X
λ_k	Constant failure rate per time unit of the processor u_k
num_i	Number of replicas of the task n_i
$NR(G)$	Total number of the replicas of the application G
$lb(n_i)$	Lower bound on number of replicas of the task n_i
n_i^x	xth replica of the task n_i
$u_{pr(n_i^x)}$	Assigned processor of the replica n_i^x
$R(n_i, u_k)$	Reliability of the task n_i on the processor u_k
$R(n_i)$	Reliability of the task n_i
$R(G)$	Reliability of the application G
$R_{\mathrm{seq}}(G)$	Reliability requirement of the application G
$R_{\mathrm{seq}}(n_i)$	Sub-reliability requirement of the task n_i
$R_{\mathrm{up\_seq}}(n_i)$	Upper bound on reliability requirement of the task n_i

As mentioned earlier, a parallel application running on processors is represented by a DAG G=(N, W, M, C) with known values.

(1) N represents a set of nodes in G, and each node $n_i \in N$ is a task with different execution time values on different processors. In addition, task executions of a given application are assumed to be non-preemptive which is possible in many systems [8], [14]. $pred(n_i)$ is the set of immediate predecessor tasks of n_i, while $succ(n_i)$ is the set of immediate successor tasks of n_i. Tasks without predecessor tasks are denoted by n_{entry}; and tasks with no successor tasks are denoted by n_{exit}. If an application has multiple entry or multiple exit tasks, then a dummy entry or exit task with zero-weight dependencies is added to the graph. W is an $|N| \times |U|$ matrix in which $w_{i,k}$ denotes the execution time of n_i running on u_k.

(2) M is a set of communication edges, and each edge $m_{i,j} \in M$ represents a communication from n_i to n_j. Accordingly, $c_{i,j} \in C$ represents the communication time of $m_{i,j}$ if n_i and n_j are assigned to different processors because two tasks with immediate precedence constraints need to exchange messages. When both tasks n_i to n_j are allocated to the same processor, $c_{i,j}$ becomes zero because we assume that the intra-processor communication cost is negligible [14], [15].

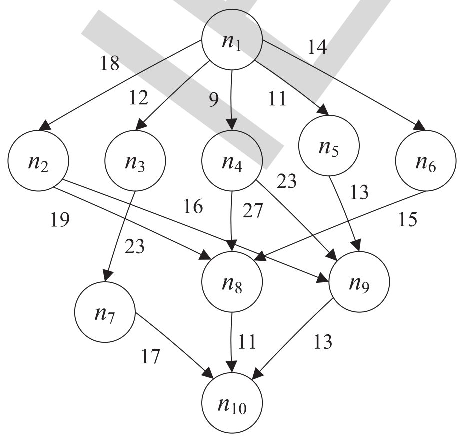


Fig. 2. Motivating example of a DAG-based parallel application with 10 tasks [9], [10], [11], [12].

TABLE 2
Execution Time Values of Tasks on Different Processors of the Motivating Parallel Application [9], [10], [11]

Task	u_1	u_2	u_3
n_1	14	16	9
n_2	13	19	18
n_3	11	13	19
n_4	13	8	17
n_5	12	13	10
n_6	13	16	9
n_7	7	15	11
n_8	5	11	14
n_9	18	12	20
n_{10}	21	7	16

Fig. 2 shows a motivating parallel application with tasks and messages [9], [10], [11], [12]. The example shows 10 tasks executed on 3 processors $\{u_1, u_2, u_3\}$. The weight 18 of the edge between n_1 and n_2 represents communication time, denoted by $c_{1,2}$ if n_1 and n_2 are not assigned to the same processor.

Table 2 is the execution time matrix $|N| \times |U|$ of tasks on different processors of the motivating parallel application. For example, the weight 14 of n_1 and u_1 in Table 2 represents execution time of n_1 on u_1, denoted by $w_{1,1}$=14. We can see that the same task has different execution time values on different processors due to the heterogeneity of the processors.

The motivating example will be used to explain the MaxRe, RR, and the proposed LBR, ERRM, and HRRM algorithms in the paper.

3.2 Reliability Model

There are two major types of failures: transient failure (also called random hardware failure) and permanent failure. Once a permanent failure occurs, the processor cannot be restored unless by replacement. The transient failure appears for a short time and disappear without damage to processors. Therefore, this paper mainly takes the transient failures into account for our research. In general, the occurrence of transient failure for a task in a DAG-based application follows the Poisson distribution [14], [15], [19], [31], [36]. The reliability of an event in unit time t is denoted by

$$R(t) = e^{-\lambda t},$$

where λ is the *constant failure rate per time unit* for a processor. We use λ_k to represent the constant failure rate per time unit of the processor u_k. The reliability of n_i executed on u_k in its execution time is denoted by

$$R(n_i, u_k) = e^{-\lambda_k w_{i,k}}, \tag{1}$$

and the failure probability for n_i without using the active replication is

$$1 - R(n_i, u_k) = 1 - e^{-\lambda_k w_{i,k}}. \tag{2}$$

However, each task has a number of replicas with the active replication. We define num_i ($num_i \leqslant |U|$) as the number of replicas of n_i. Hence, the replica set of n_i is $\{n_i^1, n_i^2, \ldots, n_i^{num_i}\}$ where n_i^1 is the primary and others are backups. As long as one replica of n_i is successfully completed, then we can recognize that there is no occurrence of failure for n_i, and the reliability of n_i is updated to

$$R(n_i) = 1 - \prod_{x=1}^{num_i} \left(1 - R\left(n_i^x, u_{pr(n_i^x)}\right)\right), \tag{3}$$

where $u_{pr(n_i^x)}$ represents the assigned processor of n_i^x. The difference between $R(n_i, u_k)$ and $R(n_i)$ is below: $R(n_i, u_k)$ is the value before task replication, whereas $R(n_i)$ is the value after task replication.

The reliability of the parallel application with precedence-constrained tasks should be [14]

$$R(G) = \prod_{n_i \in N} R(n_i). \tag{4}$$

In [15], both communication and computation failures are considered; however, some communication networks themselves provide fault-tolerance. For instance, routing information protocol (RIP) and open shortest path first (OSPF) are designed to reroute packets to ensure that they reach their destination [37]. Therefore, similar to some studies [14], [35], [38], this study only considers processor failure and excludes communication failure (i.e., the communication is assumed to be reliable in this study). In addition, we mainly focus on the redundancy minimization of tasks, which is not directly related to communication.

3.3 Problem Statement

As discussed in Section 1, any application cannot be 100 percent reliable, but if the system can satisfy application's reliability requirement, then the application is considered reliable. The problem addressed in this study can be formally described as follows. Assume that we are given a parallel application G and a heterogeneous processor set U. The problem is to assign replicas and corresponding processors for each task, while minimizing the number of replicas and ensuring that the obtained reliability of the application $R(G)$ satisfies the application's reliability requirement $R_{seq}(G)$. The formal description is to find the replicas and processor assignments of all tasks to minimize

$$NR(G) = \sum_{n_i \in N} num_i, \tag{5}$$

subject to

$$R(G) = \prod_{n_i \in N} R(n_i) \geqslant R_{\text{req}}(G),$$

for $\forall i : 1 \leqslant i \leqslant |N|$.

TABLE 3
Upward Rank Values for Tasks of the Motivating Parallel Application

Task	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}
$rank_{\text{u}}(n_i)$	108	77	80	80	69	63.3	42.7	35.7	44.3	14.7

4 State-of-the-Art Approaches

4.1 Task Prioritizing

A fault-tolerant scheduling algorithm generally consists of three steps: 1) task prioritizing, 2) processor selection, and 3) task execution. Therefore, we should first compute the task priority before processor selection. Similar to state-of-the-art studies [14], [15], this study uses the famous upward rank value ($rank_{\text{u}}$) of a task (Eq. (6)) as the task priority standard. In this case, the tasks are ordered by descending order of $rank_{\text{u}}$, which are obtained by Eq. (6) [9], as follows:

$$rank_{\text{u}}(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} \{c_{i,j} + rank_{\text{u}}(n_j)\}, \tag{6}$$

in which $\overline{w_i}$ represents the average execution time of task n_i and is calculated as follows:

$$\overline{w_i} = \left(\sum_{k=1}^{|U|} w_{i,k}\right) \bigg/ |U|.$$

Table 3 shows the upward rank values of all the tasks in Fig. 2. Note that only if all the predecessors of n_i have been assigned, will n_i prepare to be assigned. Assume that two tasks n_i and n_j satisfy $rank_{\text{u}}(n_i) > rank_{\text{u}}(n_j)$; if no precedence constraint exists between n_i and n_j, n_i does not necessarily take precedence for n_j to be assigned. Therefore, the task assignment order in G is $\{n_1, n_3, n_4, n_2, n_5, n_6, n_9, n_7, n_8, n_{10}\}$.

4.2 Existing MaxRe Algorithm

As the application reliability is the product of all the task reliability values, such problem is usually solved by transferring application's reliability requirement to the sub-reliability requirements of tasks [14], [15], [39]. In the MaxRe algorithm [14], the sub-reliability requirement for each task is calculated by

$$R_{\text{req}}(n_i) = \sqrt[|N|]{R_{\text{req}}(G)}. \tag{7}$$

If the sub-reliability requirement of each task can be satisfied by active replication below

$$R(n_i) \geqslant R_{\text{req}}(n_i),$$

then obviously the application's reliability requirement can be satisfied. Therefore, the main idea of the MaxRe algorithm is to iteratively select the replica n_i^x and processor $u_{pr(n_i^x)}$ with the maximum $R(n_i^x, u_{pr(n_i^x)})$ until the actual reliability value is larger than or equal to the sub-reliability requirement of the task, namely,

$$R(n_i) \geqslant R_{\text{req}}(n_i).$$

Moreover, this policy was also employed by the authors in [39].

TABLE 4
Task Assignment of the Motivating Parallel Application Using the MaxRe Algorithm

n_i	$R_{\rm req}(n_i)$	$R(n_i,u_1)$	$R(n_i,u_2)$	$R(n_i,u_3)$	num_i	$R(n_i)$
n_1	0.99383156	**0.98609754**	0.97628571	**0.98393051**	2	0.99977659
n_3	0.99383156	**0.98906028**	**0.98068890**	0.96637821	2	0.99978874
n_4	0.99383156	**0.98708414**	**0.98807171**	0.96986344	2	0.99984594
n_2	0.99383156	**0.98708414**	**0.97190229**	0.96811926	2	0.99963709
n_5	0.99383156	**0.98807171**	0.98068890	**0.98216103**	2	0.99978721
n_6	0.99383156	**0.98708414**	0.97628571	**0.98393051**	2	0.99979245
n_9	0.99383156	**0.98216103**	**0.98216103**	0.96464029	2	0.99968177
n_7	0.99383156	**0.99302444**	0.97775124	**0.98039473**	2	0.99986324
n_8	0.99383156	**0.99501248**	0.98363538	0.97511487	1	0.99501248
n_{10}	0.99383156	**0.97921896**	**0.98955493**	0.97161077	2	0.99978294
$NR(G)=19$, $R(G)=0.99298048$						

TABLE 5
Task Assignment of the Motivating Parallel Application Using the RR Algorithm

n_i	$R_{\rm req}(n_i)$	$R(n_i,u_1)$	$R(n_i,u_2)$	$R(n_i,u_3)$	num_i	$R(n_i)$
n_1	0.99383156	**0.98609754**	0.97628571	**0.98393051**	2	0.99977659
n_3	0.99317319	**0.98906028**	**0.98068890**	0.96637821	2	0.99978874
n_4	0.99234932	**0.98708414**	**0.98807171**	0.96986344	2	0.99984594
n_2	0.99128298	**0.98708414**	**0.97190229**	0.96811926	2	0.99963709
n_5	0.98989744	**0.98807171**	0.98068890	**0.98216103**	2	0.99978721
n_6	0.98793125	**0.98708414**	0.97628571	**0.98393051**	2	0.99979245
n_9	0.98498801	**0.98216103**	**0.98216103**	0.96464029	2	0.99968177
n_7	0.98013824	**0.99302444**	0.97775124	0.98039473	1	0.99302444
n_8	0.97511487	**0.99501248**	0.98363538	0.97511487	1	0.99501248
n_{10}	0.97161077	0.97921896	**0.98955493**	0.97161077	1	0.98955493
$NR(G)=17$, $R(G)=0.97609982$						

Example 1. Assume that the constant failure rates for three processors are $\lambda_1=0.0010$, $\lambda_2=0.0015$, and $\lambda_3=0.0018$, respectively. Moreover, assume that the reliability requirement of the parallel application in Fig. 2 is $R_{\rm seq}(G)=0.94$. Note that the above values are not the representatives of a real deployment, but are used to explain the example clearly.

Table 4 shows the task assignment for each task of the motivating parallel application using the MaxRe algorithm. Each row shows the selected processors (denoted with bold text) and corresponding reliability values. For example, the sub-reliability requirement of n_1 is $R_{\rm req}(n_1)=\sqrt[10]{0.94}=0.99383156$; to satisfy the sub-reliability requirement, MaxRe selects the processors u_1 and u_3 with the maximum and second maximum reliability values, respectively (i.e., $num_1=2$). Then, the actual reliability value of n_1 is 0.99977659, which is calculated by Eq. (3). The remaining tasks use the same pattern with n_1. Finally, the number of replicas are 19 and the actual reliability value of the application G is 0.99298048, which are calculated by Eqs. (5) and (4), respectively.

4.3 Existing RR Algorithm

Obviously, the main limitation of the MaxRe algorithm is that the sub-reliability requirements of all tasks are equal and high, such that it needs more replicas with extra redundancy to satisfy the sub-reliability requirement of each task. To solve such problem, the authors presented the RR algorithm to lower down the sub-reliability requirement of tasks while still satisfying the application's reliability requirement [15] as follows.

First, the sub-reliability requirement for the entry task is still calculated by

$$R_{\rm req}(n_1)=\sqrt[|N|]{R_{\rm req}(G)}.$$

Second, for the rest of tasks (i.e., non-entry tasks), unlike prior MaxRe algorithm [14], sub-reliability requirements in the RR algorithm are calculated continuously based on the actual reliability achieved by previous allocations

$$R_{\rm req}(n_{seq(j)})=\sqrt[|N|-j+1]{\frac{R_{\rm req}(G)}{\prod_{x=1}^{j-1}R(n_{seq(x)})}}, \tag{8}$$

where $n_{seq(j)}$ represents the jth assigned task. Clearly, such single improvement can reduce the sub-reliability requirements of non-entry tasks.

Example 2. The same parameter values ($\lambda_1=0.0010$, $\lambda_2=0.0015$, $\lambda_3=0.0018$, and $R_{\rm seq}(G)=0.94$) with Example 1 are used. Table 5 shows the task assignment for each task of the motivating parallel application using the RR algorithm. Each row shows the selected processors (denoted with bold text) and corresponding reliability values. The sub-reliability requirement and task assignment of n_1 using the RR algorithm is similar to the MaxRe algorithm. However, the remaining tasks are different. For example, as the actual reliability value for n_1 is 0.99977659, then the sub-reliability requirement for n_3 should be $\sqrt[9]{\frac{0.94}{0.99977659}}=0.99317319$. When assigning n_7 and n_8, the sub-reliability requirements are reduced to 0.98013824 and 0.97161077, respectively. That is, only one replica for each of n_7 and n_8 will be able to satisfy individual sub-reliability requirements. Finally, the number of replicas and the actual reliability value of the application G are 17 and 0.97609982 (calculated by Eqs. (5) and (4)), respectively, which still satisfy application's reliability requirement, but their values are less than those obtained with the MaxRe algorithm.

5 Enough Replication for Redundancy Minimization

Although the RR algorithm can reduce the sub-reliability requirements of tasks, the reduction ranges of tasks near the entry task are much lower than those of the tasks near the exit task. That is, the actual sub-reliability requirements show unfairness among tasks, such that the RR algorithm still requires unnecessary redundancy to satisfy application's reliability requirement. To further reduce redundancy, we first present good enough replication approach in this section, and then propose a heuristic replication approach in the next section.

5.1 Lower Bound on Redundancy

Considering that application reliability is the product of all task reliability values, the reliability value of each task should be higher than or equal to $R_{\rm req}(G)$; otherwise, if one task has

$R(n_i) < R_{\text{req}}(G)$, then no matter how many replicas for any other tasks, $R_{\text{req}}(G)$ cannot be satisfied. Therefore, the lower bound on reliability requirement of the task n_i is

$$R_{\text{lb\_req}}(n_i) = R_{\text{req}}(G). \tag{9}$$

In this way, there should be a lower bound on the number of replicas for each task that satisfies

$$R(n_i) \geqslant R_{\text{lb\_req}}(n_i).$$

In other words, we can determine the lower bound on the number of replicas $lb(n_i)$ for task n_i to satisfy

$$1 - \prod_{x=1}^{lb(n_i)} (1 - R(n_i^x, u_{pr(n_i^x)})) \geq R_{\text{lb\_req}}(n_i), \tag{10}$$

according to Eq. (3).

We use the following steps to select the replica and the corresponding processor with the minimum number of replicas.

(1) Calculate the $R(n_i, u_k)$ of each task on all available processors (if a replica of n_i has been assigned to the processor, then this processor is unavailable for n_i; otherwise, it is available for n_i).
(2) To minimize the number of replicas, select the replica n_i^x of the task n_i and the corresponding processor $u_{pr(n_i^x)}$ with the maximum $R(n_i^x, u_{pr(n_i^x)})$.
(3) Repeat Steps (1) and (2) until Eq. (10) is satisfied.

5.2 The LBR Algorithm

On the basis of the above steps, we propose the lower bound on redundancy (LBR) algorithm (Algorithm 1) to generate the lower bound on the number of the replicas of each task.

Algorithm 1. The LBR Algorithm

Input: $G = (N, W, M, C)$, U, $R_{\text{req}}(G)$
Output: $R(G)$, $NR(G)$ and its related values
1: **for** $(i = 1; i <= |N|; i++)$ **do**
2: $R_{\text{lb\_req}}(n_i) \leftarrow R(G)$;
3: $num_i = 0$;
4: $R(n_i) = 0$; // initial value is 0
5: **while** $(R(n_i) < R_{\text{lb\_req}}(n_i))$ **do**
6: Calculate $R(n_i, u_k)$ for the task n_i on all available processors using Eq. (1);
7: Select replica n_i^x and the processor $u_{pr(n_i^x)}$ with the maximum reliability value $R(n_i^x, u_{pr(n_i^x)})$;
8: num_i++;
9: Calculate $R(n_i)$ using Eq. (3);
10: **end while**
11: Calculate $NR(G)$ using Eq. (5);
12: Calculate $R(G)$ using Eq. (4);
13: **end for**

The core idea of the LBR algorithm is that each task iteratively selects the replica and available processor with the maximum reliability value $R(n_i^x, u_{pr(n_i^x)})$ for each task until the task's lower bound on reliability requirement is satisfied. The details are explained as follows:

TABLE 6
Task Assignment of the Motivating Parallel Application Using the LBR Algorithm

n_i	$R_{\text{req}}(n_i)$	$R(n_i, u_1)$	$R(n_i, u_2)$	$R(n_i, u_3)$	num_i	$R(n_i)$
n_1	0.94	**0.98609754**	0.97628571	0.98393051	1	0.98609754
n_3	0.94	**0.98906028**	0.98068890	0.96637821	1	0.98906028
n_4	0.94	0.98708414	**0.98807171**	0.96986344	1	0.98807171
n_2	0.94	**0.98708414**	0.97190229	0.96811926	1	0.98708414
n_5	0.94	**0.98807171**	0.98068890	0.98216103	1	0.98807171
n_6	0.94	**0.98708414**	0.97628571	0.98393051	1	0.98708414
n_9	0.94	**0.98216103**	0.98216103	0.96464029	1	0.98216103
n_7	0.94	**0.99302444**	0.97775124	0.98039473	1	0.99302444
n_8	0.94	**0.99501248**	0.98363538	0.97511487	1	0.99501248
n_{10}	0.94	0.97921896	**0.98955493**	0.97161077	1	0.98955493
$NR(G) = 10$, $R(G) = 0.89092057$						

(1) In Line 2, LBR has obtained the lower bound on reliability requirement of the current task before it prepares to be assigned.
(2) In Lines 5-10, LBR iteratively selects the replica and available processor for each task with the maximum reliability value until the task's lower bound on reliability requirement is satisfied. Specifically, the following details are made: 1) Line 5 compares the actual reliability value and lower bound on reliability requirement of the current task; 2) Lines 6-7 calculate and select the replica and available processor with the maximum reliability value for the current task; and 3) Line 9 calculates the actual reliability value of the current task.
(3) In Lines 11-12, LBR calculates the final number of replicas and the actual reliability value of the application, respectively.

5.3 Time Complexity of the LBR Algorithm

The time complexity of the LBR algorithm is analyzed as follows:

(1) Calculating the reliability of the application must traverse all tasks, which can be done within O($|N|$) time (Lines 1-13).
(2) The total number of replicas for each task must be lower or equal to the number of processors, which can be done within O($|U|$) time (Lines 5-10).
(3) Selecting the replica and available processor with the maximum reliability value for the current task must traverse all processors, which can be done in O(log$|U|$) time (Line 7).

Thus, the time complexity of the LBR algorithm is O($|N| \times |U| \times \log|U|$).

5.4 Example of the LBR Algorithm

Example 3. The same parameter values ($\lambda_1 = 0.0010$, $\lambda_2 = 0.0015$, $\lambda_3 = 0.0018$, and $R_{\text{seq}}(G) = 0.94$) with aforementioned examples are used. Table 6 lists the replicas, selected processor, and reliability value of each task (denoted with bold text). We find that the reliability value of each task is higher than the application's reliability requirement of 0.94. However, the current obtained reliability value of the parallel application is only $R(G) = 0.89092057$ (calculated by Eq. (4)), which is much lower than 0.94 (application's reliability requirement). Hence, application's reliability requirement is not satisfied by merely using the LBR algorithm.

5.5 Enough Replication

Considering that all the tasks merely satisfy $R(n_i) \geq R_{\text{lb\_req}}(n_i)$ by using the LBR algorithm (Algorithm 1), we should add more new replicas for tasks to satisfy application's reliability requirement. However, choosing the remaining replicas is a complex work, because different replicas of different tasks may cause different reliability values on different processors.

Given that the current number of replicas for n_i is $h = num_i$ and the application reliability is $R(G)$, if a new replica n_i^{h+1} is assigned to the processor $u_k = u_{pr(n_i^{h+1})}$ for n_i, then the number of replicas is changed to $h+1$ and the new task reliability is changed to

$$R_{\text{new}}(n_i) = 1 - \prod_{x=1}^{h+1} (1 - R(n_i^x, u_{pr(n_i^x)})). \tag{11}$$

Then, the application reliability is enhanced because of the reliability enhancement of n_i and is changed to

$$R^i(G) = R_{\text{new}}(n_i) \times \prod_{n_j \in N, i \neq j} R(n_j). \tag{12}$$

To minimize the number of replicas for each task, we use the following steps to obtain enough minimum redundancy of the application.

(1) Each available task (if the replicas of a task have been assigned to all the processors, then this task is unavailable; otherwise, a task is available) is assumed to be replicated once on an available processor with the maximum $R(n_i, u_k)$ (Eq. (1)), and the new task sub-reliability is changed to $R_{\text{new}}(n_i)$ (Eq. (11)).

(2) Calculate the application reliability $R^i(G)$ because of the reliability enhancement of each task (Eq. (12)).

(3) Select the replica n_i^x and corresponding processor $u_{pr(n_i^x)}$ that generate the maximum $R^i(G)$ from the generated replicas in Step 2), namely,

$$R^i(G) = \max\left\{R^1(G), R^2(G), \ldots, R^{|N|}(G)\right\}. \tag{13}$$

(4) Repeat Steps (1), (2), and (3) until application's reliability requirement (Eq. (4)) is satisfied.

5.6 The ERRM Algorithm

In this section, we propose the ERRM algorithm to minimize redundancy to satisfy application's reliability requirement, and describe the steps in Algorithm 2.

The core idea of the ERRM algorithm is that all the tasks are first assumed to be replicated once on an available processor with the maximum reliability values; then ERRM selects the replica n_s^x and corresponding processor $u_{pr(n_s^x)}$ that generate the maximum application reliability value $R^s(G)$ until application's reliability requirement is satisfied in the iterative replication process. The details are explained as follows:

(1) In Line 1, ERRM calls the LBR algorithm (Algorithm 1) to obtain the initial reliability $R(G)$ and related values.

(2) In Lines 2-11, ERRM iteratively selects the replica and available processor that generate the maximum application reliability value until application's reliability requirement is satisfied. Specifically, the following details are made: 1) Line 2 compares the actual reliability value and the reliability requirement of the application; 2) Lines 3-7 pre-replicate all tasks once on an available processor with the maximum reliability values; 3) Line 8 selects the replica and corresponding processor that generate the maximum application reliability value; and 4) Line 10 updates the application's reliability value.

(3) In Line 13, ERRM calculates the final number of replicas of the application.

Algorithm 2. The ERRM Algorithm

Input: $G = (N, W, M, C)$, U, $R_{\text{req}}(G)$
Output: $R(G)$, $NR(G)$ and its related values
1: Call the LBR algorithm (Algorithm 1) to obtain the initial reliability $R(G)$ and related values;
2: **while** $(R(G) < R_{\text{req}}(G))$ **do**
3: **for** $(i = 1; i <= |N|; i++)$ **do**
4: Pre-replicated the replica of n_i on an available processor with the maximum reliability value $R(n_i, u_k)$;
5: Update the task's sub-reliability value to $R_{\text{new}}(n_i)$ (Eq. (11));
6: Calculate the application reliability $R^i(G)$ after the reliability enhancement of n_i (Eq. (12));
7: **end for**
8: Select the replica n_s^x and corresponding processor $u_{pr(n_s^x)}$ that generate the maximum application reliability value $R^s(G)$ (Eq. (13));
9: num_i++;
10: $R(n_i) \leftarrow R_{\text{new}}(n_i)$;
11: $R(G) \leftarrow R^i(G)$;
12: **end while**
13: Calculate $NR(G)$ using Eq. (5);

5.7 Time Complexity of the ERRM Algorithm

The time complexity of the ERRM algorithm is analyzed as follows:

(1) The maximum number of iterative replication process is $|N| \times |U|$, which can be done within $O(|N| \times |U|)$ time (Lines 2-12).

(2) Each task must be assumed to be replicated once on an available processor, which can be done in $O(|N|)$ time (Lines 3-7).

(3) Selecting the replica and available processor with the maximum reliability value must traverse all processors, which can be done in $O(\log|U|)$ time (Line 4).

(4) Updating the task's new sub-reliability value can be done in $O(|U|)$ time (Line 5).

(5) Calculating the application's new reliability value can be done in $O(|N|)$ time (Line 6).

(6) Obtaining the maximum application reliability value can be done in $O(|N|)$ time (Line 8).

Considering that (3), (4), and (5) are not nested in the algorithm, the time complexity of the ERRM algorithm is $O(|N|^2 \times |U|^2 + |N|^3 \times |U|)$.

TABLE 7
Selected Processor and Reliability Pairs (Denoted with Underline Text) of Each Task in Each Step of the Motivating Parallel Application Using the ERRM Algorithm

Step	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}
(1)	$(u_3, 0.9033)$	$(u_2, 0.9023)$	$(u_2, 0.9006)$	$(u_1, 0.9015)$	$(u_3, 0.9015)$	$(u_3, 0.9024)$	$(u_3, 0.89714)$	$(u_2, 0.8953)$	$\underline{(u_2, 0.9068)}$	$(p_1, 0.9001)$
(2)	$\underline{(u_3, 0.9194)}$	$(u_2, 0.9183)$	$(u_2, 0.9166)$	$(u_1, 0.9176)$	$(u_3, 0.9176)$	$(u_3, 0.9185)$	$(u_2, 0.9131)$	$(u_2, 0.9113)$	$(u_3, 0.9071)$	$(u_1, 0.9162)$
(3)	$(u_2, 0.9196)$	$(u_2, 0.9311)$	$(u_2, 0.9294)$	$(u_1, 0.9303)$	$(u_3, 0.9303)$	$\underline{(u_3, 0.9312)}$	$(u_3, 0.9257)$	$(u_2, 0.9239)$	$(u_3, 0.9197)$	$(u_1, 0.9289)$
(4)	$(u_2, 0.9314)$	$\underline{(u_2, 0.9431)}$	$(u_2, 0.9413)$	$(u_1, 0.9423)$	$(u_3, 0.9423)$	$(u_2, 0.9314)$	$(u_3, 0.9376)$	$(u_2, 0.9358)$	$(u_3, 0.9315)$	$(u_1, 0.9409)$

Considering that the time complexity of the LBR algorithm (i.e., O($|N| \times |U| \times \log|U|$)) is less than that of the ERRM algorithm, using the LBR algorithm in advance can improve the efficiency of the replication compared with only using the ERRM algorithm. The reason is that the LBR algorithm can obtain an initial reliability value greater than zero, such that the number of iterative process of the ERRM algorithm can be reduced. Considering the motivating example, the reliability value obtained is 0.89092057, shown in Table 6, then the initial reliability value is not 0, but 0.89092057. Compared to starting from 0, 0.89092057 is close to the actual reliability requirement of 0.93.

5.8 Example of the ERRM Algorithm

Example 4. The same parameter values ($\lambda_1 = 0.0010$, $\lambda_2 = 0.0015$, $\lambda_3 = 0.0018$, and $R_{\text{seq}}(G) = 0.94$) with aforementioned examples are used. Table 7 lists the selected processor and reliability pairs of each task in each step by using Algorithm 2, where the underlined values indicate those that have the maximum $R_{\text{new}}(n_i)$ (Eq. (11)) and $R^{\text{s}}(G)$ (Eq. (13)), and the replica is selected to enhance the reliability of the application in each step. For example, in Step (1), n_9 and u_2 are selected, because they can generate the maximum value of 0.9068. In Step (4), the reliability value is larger than or equal to application's reliability requirement 0.94. Hence, application's reliability requirement is satisfied, and the replication process successfully ends.

Table 8 lists the final replicas, selected processor, and reliability value for each task of the parallel application in Fig. 2. We find that the final reliability value of each task is larger than or equal to 0.94. Moreover, the current reliability value is $R(G) = 0.94307237$ (calculated by Eq. (4)), which is larger than 0.94. Hence, application's reliability requirement is satisfied, and the application proves reliable in this situation. Meanwhile, the final resource consumption is $NR(G) = 14$ (Calculated by Eq. (5)).

TABLE 8
Task Assignment of the Application in Fig. 2 Using the ERRM Algorithm

n_i	$R(n_i, u_1)$	$R(n_i, u_2)$	$R(n_i, u_3)$	num_i	$R(n_i)$
n_1	**0.98609754**	0.97628571	$\underline{0.98393051}$	2	0.99977659
n_3	**0.98906028**	0.98068890	$\underline{0.96637821}$	1	0.98906028
n_4	0.98708414	**0.98807171**	0.96986344	1	0.98807171
n_2	**0.98708414**	$\underline{0.97190229}$	0.99963709	2	0.98708414
n_5	**0.98807171**	$\underline{0.98068890}$	0.98216103	1	0.98807171
n_6	**0.98708414**	0.97628571	$\underline{0.98393051}$	2	0.99979245
n_9	**0.98216103**	$\underline{0.98216103}$	$\underline{0.96464029}$	2	0.99968177
n_7	**0.99302444**	$\underline{0.97775124}$	0.98039473	1	0.99302444
n_8	**0.99501248**	0.98363538	0.97511487	1	0.99501248
n_{10}	0.97921896	**0.98955493**	0.97161077	1	0.98955493
$NR(G) = 14$, $R(G) = 0.94307237$					

6 Heuristic Replication for Redundancy Minimization

Although the ERRM algorithm can implement enough redundancy minimization, it has high time complexity and thereby it is time-consuming for a large-scale parallel application. To reduce the redundancy of a large-scale parallel application within an acceptable computation time, this section presents a heuristic algorithm.

6.1 Upper Bound on Reliability Requirement

Although the RR algorithm can achieve more redundancy reduction than the MaxRe algorithm by recalculating the sub-reliability requirement, the redundancy reduction ranges of the tasks near the entry task is much lower than those of the tasks near the exit task (see Table 5). The main reason for the discrepancy is that unfair sub-reliability requirements among tasks are generated. In fact, the tasks that are after $n_{seq(x)}$'s allocations (i.e., unassigned tasks) can also be presupposed as assigned tasks with known reliability values.

We find that all the sub-reliability requirements of tasks using the RR algorithm do not exceed 0.99383156 (see Table 5), which is the sub-reliability requirement of each task using the MaxRe algorithm (see Table 4). Thus, we let $\sqrt[|N|]{R_{\text{req}}(G)}$ be the upper bound on task's reliability requirement, namely,

$$R_{\text{up\_req}}(n_i) = \sqrt[|N|]{R_{\text{req}}(G)}. \tag{14}$$

Then, we have the following heuristic strategy: assume that the task to be assigned is $n_{seq(j)}$ ($n_{seq(j)}$ represents the jth assigned task as mentioned earlier), then $\{n_{seq(1)}, n_{seq(2)}, \ldots, n_{seq(j-1)}\}$ represents the task set with assigned tasks, and $\{n_{seq(j+1)}, n_{seq(j+2)}, \ldots, n_{seq(|N|)}\}$ represents the task set with unassigned tasks. To ensure that the reliability of the application is satisfied at each task assignment, we presuppose that each task in $\{n_{seq(j+1)}, n_{seq(j+2)}, \ldots, n_{seq(|N|)}\}$ is assigned to the processor with reliability value on upper bound (Eq. (14)). Hence, when assigning $n_{seq(j)}$, application's reliability requirement is

$$R_{\text{req}}(G) = \prod_{x=1}^{j-1} R(n_{seq(x)}) \times R_{\text{req}}(n_{seq(j)}) \times \prod_{y=j+1}^{|N|} R_{\text{up\_req}}(n_{seq(y)}).$$

Then, the sub-reliability requirement for the task $n_{seq(j)}$ should be

$$R_{\text{req}}(n_{seq(j)}) = \frac{R_{\text{req}}(G)}{\prod_{x=1}^{j-1} R(n_{seq(x)}) \times \prod_{y=j+1}^{|N|} R_{\text{up\_req}}(n_{seq(y)})}. \tag{15}$$

6.2 The HRRM Algorithm

On the basis of the aforementioned new sub-reliability requirement calculation for each task (Eq. (15)), we present the heuristic algorithm HRRM described in Algorithm 3 to minimize redundancy and satisfy application's reliability requirement.

Algorithm 3. The HRRM Algorithm

Input: $G = (N, W, M, C)$, U, $R_{\text{req}}(G)$
Output: $R(G)$, $NR(G)$ and its related values
1: Order tasks according to a descending order of $rank_{\text{u}}(n_i, u_k)$ using Eq. (6);
2: **for** $(j = 1; j <= |N|; j++)$ **do**
3:   Calculate $R(n_{seq(j)})$ using Eq. (3);
4:   $num_{seq(j)} = 0$;
5:   $R(n_{seq(j)}) = 0$; // initial value is 0
6:   Calculate $R_{\text{req}}(n_{seq(j)})$ using Eq. (15);
7:   **while** $(R(n_{seq(j)}) < R_{\text{req}}(n_{seq(j)}))$ **do**
8:     Calculate $R(n_{seq(j)}, u_k)$ for the task $n_{seq(j)}$ on all each available processor using Eq. (1);
9:     Select replica $n^x_{seq(j)}$ and the processor $u_{pr(n^x_{seq(j)})}$ with the maximum $R(n^x_{seq(j)}, u_{pr(n^x_{seq(j)})})$;
10:     $num_{seq(j)}$++;
11:     Calculate $R(n_{seq(j)})$ using Eq. (3);
12:   **end while**
13: **end for**
14: Calculate $NR(G)$ using Eq. (5);
15: Calculate $R(G)$ using Eq. (4);

The core idea of HRRM is that the reliability requirement of the application is transferred to the sub-reliability requirement of each task. Each task just iteratively selects the replica and available processor with the maximum reliability value until its sub-reliability requirement is satisfied. The details are explained as follows:

(1) In Line 6, HRRM has obtained the reliability requirement of the current task before it prepares to be assigned.
(2) In Lines 7-12, HRRM iteratively selects the replica and available processor with the maximum reliability value for the current task until its sub-reliability requirement is satisfied. Specifically, the following details are made: 1) Line 7 compares the actual reliability value and sub-reliability requirement of the current task; 2) Lines 8-9 calculate and select the replica and available processor with the maximum reliability value for the current task; and 3) Line 11 calculates the actual reliability value of the current task.
(3) In Lines 14-15, HRRM calculates the final number of replicas and the actual reliability value of the application, respectively.

Compared with MaxRe and RR algorithms, the main improvement of the presented HRRM is that it recalculates the sub-reliability requirement of each task based not only on its previous assignments ($\{n_{seq(1)}, n_{seq(2)}, \ldots, n_{seq(j-1)}\}$), but also on succeeding pre-assignments $\{n_{seq(j+1)}, n_{seq(j+2)}, \ldots, n_{seq(|N|)}\}$, whereas MaxRe algorithm has a fixed and equal sub-reliability requirements for all tasks and RR algorithm is merely based on previous assignments.

TABLE 9
Task Assignment of the Motivating Parallel Application Using the HRRM Algorithm

n_i	$R_{\text{req}}(n_i)$	$R(n_i, u_1)$	$R(n_i, u_2)$	$R(n_i, u_3)$	num_i	$R(n_i)$
n_1	0.99383156	**0.98609754**	0.97628571	**0.98393051**	2	0.99977659
n_3	0.98792188	**0.98906028**	0.98068890	0.96637821	1	0.98906028
n_4	0.99268768	**0.98708414**	**0.98807171**	0.96986344	2	0.99984594
n_2	0.98671636	**0.98708414**	0.97190229	0.96811926	1	0.98708414
n_5	0.99346128	**0.98807171**	0.98068890	**0.98216103**	2	0.99978721
n_6	0.98754331	**0.98708414**	0.97628571	**0.98393051**	2	0.99979245
n_9	0.98165546	**0.98216103**	0.98216103	0.96464029	1	0.98216103
n_7	0.99331998	**0.99302444**	0.97775124	**0.98039473**	2	0.99986324
n_8	0.98732777	**0.99501248**	0.98363538	0.97511487	1	0.99501248
n_{10}	0.98615598	0.97921896	**0.98955493**	0.97161077	1	0.98955493
$NR(G) = 15$, $R(G) = 0.94323987$						

6.3 Time Complexity of the HRRM Algorithm

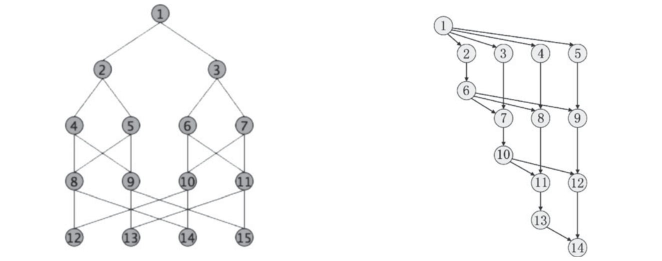
The time complexity of the HRRM algorithm is analyzed as follows:

(1) Calculating the reliability of the application must traverse all tasks, which can be done within $O(|N|)$ time (Lines 2-13).
(2) Calculating the sub-reliability requirement of the current task must traverse all tasks, which can be done within $O(|N|)$ time (Line 6).
(3) The number of replicas must be lower or equal to the number of processors, which can be done within $O(|U|)$ time (Lines 7-12).
(4) Calculating the reliability value of the current task must traverse all assigned processors, which can be done in $O(|U|)$ time (Line 11)

Considering that (2) and (3) are not nested in the algorithm, the time complexity of the HRRM algorithm is $O(|N|^2 + |N| \times |U|^2)$, which is similar to those of MaxRe and RR algorithms. Thus, HRRM implements efficient fault-tolerance without increasing time complexity.

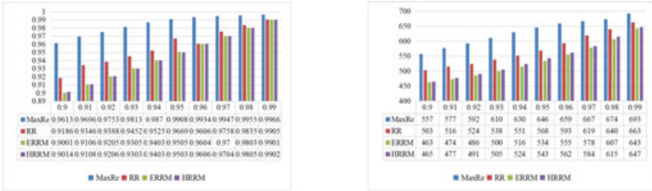
6.4 Example of the HRRM Algorithm

Example 5. The same parameter values ($\lambda_1 = 0.0010$, $\lambda_2 = 0.0015$, $\lambda_3 = 0.0018$, and $R_{\text{seq}}(G) = 0.94$) with aforementioned examples are used. Table 9 shows the task assignment for each task of the motivating parallel application using HRRM algorithm. Each row shows the selected processors (in red) and corresponding reliability values. The sub-reliability requirement and task assignment of n_1 using HRRM algorithm is similar to those using MaxRe and RR algorithms. However, the remaining tasks are different. For example, when assigning n_3, the actual reliability value for n_1 is 0.99977659, and succeeding pre-assignments with reliability requirements are $\sqrt[10]{0.94} = 0.99383156$, then the sub-reliability requirement for n_3 should be $\frac{0.94}{0.99977659 \times 0.99383156^8} = 0.98792188$. Compared with the RR algorithm, an obvious improvement for the HRRM algorithm is that it shows relative fair reliability requirements among tasks; furthermore, most sub-reliability requirements of tasks using HRRM are less than those using the RR algorithm. Finally, the number of replicas and the actual reliability value of the application



(a) Fast Fourier transform with ρ=4. (b) Gaussian elimination with ρ=5.

Fig. 3. Example of real parallel applications.



(a) Actual reliability values. (b) Total numbers of replicas.

Fig. 4. Results of the small-scale fast Fourier transform application on different reliability requirements (Experiment 1).

G are 15 and 0.94323987 (calculated by Eqs. (5) and (4)), respectively, which are lower than those with MaxRe and RR algorithms.

7 EXPERIMENTS

7.1 Experimental Metrics and Parameter Values

Considering that this study aims to implement redundancy minimization with replication to satisfy application's reliability requirement, performance metrics selected for comparison should be the actual reliability value and total number of replicas of the application. Meanwhile, computation time should be included from a time complexity perspective. The computation time is measured from the start time to the end time of an algorithm to schedule an application.

Algorithms compared with the proposed ERRM and HRRM algorithms are the state-of-the-art MaxRe [14] and RR [15] algorithms. MaxRe and RR algorithms address the same problem of minimizing resource redundancy of a parallel application to satisfy application's reliability requirement on heterogeneous distributed systems.

Considering that this study focuses on the design phase, the processor and application parameters used in this phase are known. In other words, these values have been obtained in the analysis phase and are as follows [15]: 10,000 s $\leqslant w_{i,k} \leqslant$ 100,000 s, 10,000 s $\leqslant c_{i,j} \leqslant$ 100,000 s, and $0.000001 \leqslant \lambda_k \leqslant 0.000009$. The aforementioned values are generated with uniform distribution.

The parallel applications will be tested on a simulated heterogeneous system based on the above real processor and application parameter values to reflect a real deployment. A main advantage of simulation is that it can greatly reduce development cost during the design phase and effectively provide certain optimization guide to the implementation phase. The simulated multiprocessor system is configured 64 heterogeneous processors by creating 64 processor objects based known parameter values using Java on a standard desktop computer with 2.6 GHz Intel CPU and 4 GB memory.

Meanwhile, real parallel applications with precedence constrained tasks, such as fast Fourier transform and Gaussian elimination applications, are widely used in distributed systems [9], [15]. The Fourier transform and Gaussian elimination application are two typical parallel applications with high and low parallelism, respectively. To verify the effectiveness and validity of the proposed algorithms, we use the two types of real parallel applications to compare the results of all the algorithms.

A new parameter ρ is used as the size of the fast Fourier transform application. The total number of tasks is $|N| = (2 \times \rho - 1) + \rho \times \log_2{}^{\rho}$, where $\rho = 2^y$ for some integer y [9]. Fig. 3a shows an example of the fast Fourier transform application with ρ=4. Notably, ρ exit tasks exist in the fast Fourier transform application with the size of ρ. To adopt the application model of this study, we add a virtual exit task, and the last ρ tasks are set as the immediate predecessor tasks of the virtual exit task. A new parameter ρ is used as the matrix size of the Gaussian elimination application, and the total number of tasks is $|N| = \frac{\rho^2+\rho-2}{2}$ [9]. Fig. 3b shows an example of the Gaussian elimination parallel application with ρ=5.

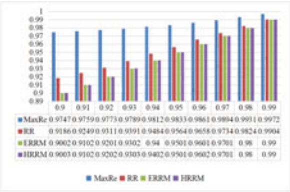
7.2 Fast Fourier Transform Application

Experiment 1. This experiment compares the actual reliability values and the total number of replicas of a small-scale fast Fourier transform application with $\rho = 32$ (i.e., $|N| = 223$) for varying reliability requirements. $R_{\text{seq}}(G)$ is changed from 0.9 to 0.99 with 0.01 increments. Note that computation time values of all the algorithms are within one second for the small-scale application and we no longer list such values in this experiment.

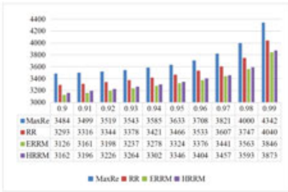
Note that the plotted values in Figs. 4a and 4b are obtained by executing one run of the algorithms for one application. Many applications with the same parameter values and scales are tested and show the same regular pattern and relatively stable results as Figs. 4a and 4b. In other words, experiments are repeatable and do not affect the consistency of the results. Therefore, the plotted values are the actual values rather than the average values during the runs.

Fig. 4a shows the actual reliability values of the small-scale fast Fourier transform application on different reliability requirements. We can see that all the algorithms can satisfy the given reliability requirements in all cases. Specifically, MaxRe generates the maximum reliability values followed by RR, HRRM, and ERRM. The overrunning reliability values (i.e., $R_{\text{seq}}(G)$-$R(G)$) reach 0.0613 and 0.0246 for MaxRe and RR, respectively. On the contrary, the overrunning reliability values are very small for HRRM (0.0001-0.0008) and ERRM (0.0001-0.0006) in all cases. Considering no additional fees will be paid for the overrunning reliability values, more resources are wasted for resource providers in using MaxRe and RR.

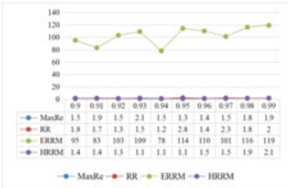
Fig. 4b shows the total number of replicas of the small-scale fast Fourier transform application on different reliability requirements. As expected, MaxRe generates the maximum numbers of replicas followed by RR, HRRM, and ERRM in all cases. The reason is that MaxRe has obtained the maximum actual reliability values followed by RR, HRRM, and ERRM in Fig. 4a, whereas optimizing reliability and



(a) Actual reliability values.



(b) Total numbers of replicas.



(c) Computation time (unit: second).

Fig. 5. Results of the large-scale fast Fourier transform application on different reliability requirements (Experiment 2).

redundancy is a bi-criteria optima problem as discussed in Section 1.2.

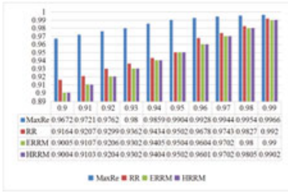
The same regular pattern for the actual reliability values is shown in Fig. 4a. As evident from Fig. 4b, the numbers of replicas using HRRM and ERRM are very similar and are much lower than those using MaxRe and RR, especially on relatively low reliability requirements. For example, when $R_{\rm seq}(G) \leqslant 0.94$, both ERRM and HRRM outperform MaxRe and RR by about 18 and 7 percent, respectively.

Experiment 2. This experiment compares the actual reliability values, the total number of replicas, and the computation time of a large-scale fast Fourier transform application with $\rho = 128$ (i.e., $|N| = 1151$) for varying reliability requirements. $R_{\rm seq}(G)$ is also changed from 0.9 to 0.99 with 0.01 increments.

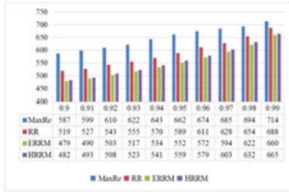
Fig. 5a shows the actual reliability values of the large-scale fast Fourier transform application on different reliability requirements. All the algorithms can satisfy the given reliability requirements in all cases. Similar to the results of the small-scale application in Fig. 4a, MaxRe still generates the maximum reliability values followed by RR, HRRM, and ERRM. Maximum differences between actual reliability and given reliability requirement are 0.0747 ($R_{\rm seq}(G) = 0.9$) and 0.0184 ($R_{\rm seq}(G) = 0.90$) for MaxRe and RR, respectively. On the contrary, in all cases the differences remain the minimum and close to application's reliability requirements using HRRM (0.0001-0.0003) and ERRM (0.0001-0.0002).

Fig. 5b shows the total number of replicas of the large-scale fast Fourier transform application on different reliability requirements. Similar to Fig. 4b in small-scale, MaxRe still generates the maximum numbers of replicas followed by RR, HRRM, and ERRM in all cases. The numbers of replicas using HRRM and ERRM are still very close and are much lower than those using MaxRe and RR in most cases.

Fig. 5c shows the computation time values of the large-scale fast Fourier transform application for reliability requirements. The values show that computation time is within 2.1 second using MaxRe, RR, and HRRM, whereas those using ERRM are 80-120 times longer. Such results indicate that ERRM is time-consuming for large-scale applications, as analyzed earlier.



(a) Actual reliability values.



(b) Total numbers of replicas.

Fig. 6. Results of the small-scale Gaussian elimination application on different reliability requirements (Experiment 3).

An interesting phenomenon is that the computation time values using ERRM for large scale applications are not increased but reduced as the application's reliability requirements increase in most cases, shown in Fig. 5c. The reason is that when using ERRM, it first calls the LBR algorithm (Algorithm 1) to obtain the initial reliability values of the application. A higher reliability requirement of the application may lead to higher initial reliability values with very short time by using LBR in these cases, such that the total computation time is not increased, but reduced with the application's reliability requirements increase.

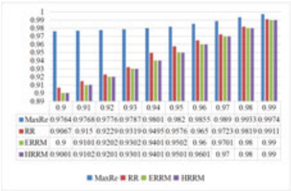
The results of Figs. 4a, and 5c show that ERRM and HRRM algorithms generate less redundancy than the state-of-the-art MaxRe and RR algorithms. Specifically, results of HRRM algorithm are very similar to those of ERRM algorithm indicating that HRRM implements approximate optimal redundancy with minimum time, whereas the enough optimal ERRM algorithm is time-consuming for large-scale parallel applications.

7.3 Gaussian Elimination Application

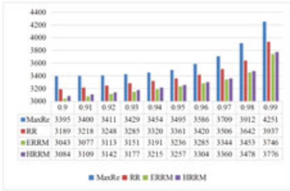
Experiment 3. This experiment compares the actual reliability values and the total number of replicas of in a small-scale Gaussian elimination application with $\rho = 21$ (i.e., $|N|$=230). The total number of task for the Gaussian elimination is similar to that of the fast Fourier transform application for varying reliability requirements. $R_{\rm seq}(G)$ is also changed from 0.9 to 0.99 with 0.01 increments. Similar to small-scale fast Fourier transform in Experiment 1, the computation time values using all the algorithms are also within one second for the small-scale Gaussian elimination. Therefore, we also no longer list such values in this experiment.

Figs. 6a and 6b show the actual reliability values and total number of replicas of the small-scale Gaussian elimination application on different reliability requirements. In general, Experiment 3 shows similar pattern and values as Experiment 1 for the total number of replicas for all the algorithms.

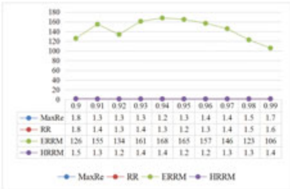
The results of Experiments 1 and 3 indicate that different parallelism degrees of applications in the same small-scale will generate similar actual reliability values and total number of replicas. In other words, parallelism degrees do not affect the scopes of actual reliability values and total number of replicas. The reason is that the reliability value of the application is the product of that of each task according to Eq. (4); considering that the number of tasks, the reliability requirement, and computation time are approximate equal, the actual reliability values and total number of replicas are also approximate equal.



(a) Actual reliability values.



(b) Total numbers of replicas.



(c) Computation time (unit: second).

Fig. 7. Results of the large-scale Gaussian elimination application on different reliability requirements (Experiment 4).

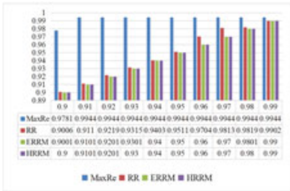
Experiment 4. This experiment compares the actual reliability values, the total number of replicas, and the computation time of a large-scale Gaussian elimination application with $\rho = 47$ (i.e., $|N| = 1,127$) for varying reliability requirements. $R_{\text{seq}}(G)$ is also changed from 0.9 to 0.99 with 0.01 increments.

Figs. 7a, 7b, and 7c show the actual reliability values, total number of replicas, and computation time of the large-scale Gaussian elimination application on different reliability requirements. Experiment 4 shows similar pattern and values as Experiment 2 in actual reliability values and total number of replicas for all the algorithms. The results of Experiments 2 and 4 further indicate that parallelism degrees do not affect the scopes of actual reliability values and total number of replicas.

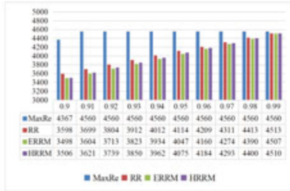
7.4 Randomly Generated Parallel Application

To extensively demonstrate the benefits of the proposed algorithms, we consider randomly generated parallel applications by the task graph generator [40]. Considering that the objective platform is heterogeneous processors, heterogeneity degrees may also affect the redundancy of application. Heterogeneity degree is easy to be implemented for randomly generated parallel applications as long as adjust the heterogeneity factor values. Randomly generated parallel applications are generated depending on the following parameters: average computation time is 50,000 ms, communication to computation ratio (CCR) is 1, and shape parameter is 1. The heterogeneity degree (factor) values belong to the scope of (0,1] in the task graph generator, where 0 and 1 represent the lowest and highest heterogeneity factors, respectively. Without loss of generality, we use large-scale randomly generated parallel application with 1,140 tasks, which are approximate equal to those of fast Fourier transform and Gaussian elimination applications in Experiments 2 and 4.

Experiment 5. This experiment compares the actual reliability values and the total number of replicas of a large-scale low-heterogeneity (with the heterogeneity factor 0.1) randomly generated parallel application with $|N| = 1,140$ for varying reliability requirements. $R_{\text{seq}}(G)$ is also changed from 0.9 to 0.99 with 0.01 increments.



(a) Actual reliability values.



(b) Total numbers of replicas.

Fig. 8. Results of the large-scale low-heterogeneity randomly generated parallel application on different reliability requirements (Experiment 5).

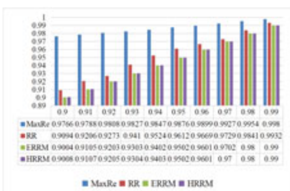
Figs. 8a and 8b show the actual reliability values and total numbers of replicas the large-scale low-heterogeneity randomly generated parallel application on different reliability requirements. It is easy to see that Experiment 5 shows similar pattern and values as Experiments 2 and 4 using all the algorithms. The main differences are as follows:

(1) The actual reliability values and total numbers of replicas obtained by MaxRe in Experiment 5 are relatively stable for different reliability requirements. The reason is that the execution time values are relative stable on the same processor for a low-heterogeneity parallel application and the reliability requirement using MaxRe is the same for all tasks, such that the values for the application do not changed much.

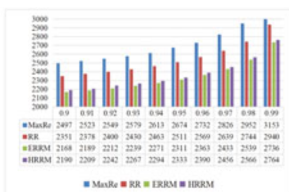
(2) The actual reliability values and total numbers of replicas obtained by RR, ERRM, and HRRM are relatively close in the same reliability requirement. The reason is still that the execution time values are relative stable on the same processor for a low-heterogeneity parallel application.

Experiment 6. This experiment compares the actual reliability values and the total number of replicas of a large-scale high-heterogeneity (with the heterogeneity factor 1) randomly generated parallel application with $|N| = 1140$ for varying reliability requirements. $R_{\text{seq}}(G)$ is also changed from 0.9 to 0.99 with 0.01 increments.

Figs. 9a and 9b show the actual reliability values and total numbers of replicas the large-scale high-heterogeneity randomly generated parallel application on different reliability requirements. It is easy to see that Experiment 6 shows similar pattern and values as Experiment 5 using all the algorithms. The main difference is that the high-heterogeneity application needs fewer replicas than the low-heterogeneity application. The total numbers of replicas for the former is only 60 percent of those for the latter using all the algorithms. The reason is that the actual reliability values for a task on different processors change much in a high-heterogeneity application, and these algorithms tend to choose the processor



(a) Actual reliability values.



(b) Total numbers of replicas.

Fig. 9. Results of the large-scale high-heterogeneity randomly generated parallel application on different reliability requirements (Experiment 6).

with the maximum reliability value for each task replication. Moreover, different from the low-heterogeneity application where the total numbers of replicas obtained by RR, ERRM, and HRRM are relatively close, ERRM and HRRM generate much less replicas than RR for the high-heterogeneity application. The reason is still that the actual reliability values for a task on different processors change much.

7.5 Summary of Experiments

Based on the above experimental results, summarizations are as follows.

(1) The proposed redundancy minimization algorithms, ERRM and HRRM, can generate less redundancy than the state-of-the-art MaxRe and RR algorithm at different scales, parallelism degrees, and heterogeneity degrees.
(2) Results of the HRRM algorithm are very similar to those of the ERRM algorithm. HRRM implements approximate optimal redundancy with minimum computation time, whereas the enough optimal ERRM algorithm is time-consuming for large-scale parallel applications.
(3) According to the analysis of the number of active processors, parallelism degrees do not affect the scopes of reliability values and total number of replicas for different types of applications in the same-scale.
(4) If the parallel application is small, then ERRM can be utilized to minimize redundancy; otherwise HRRM is the preferred alternative for reducing redundancy with minimum computation time.
(5) RR, ERRM, and HRRM obtain relatively close numbers of replicas for a low-heterogeneity application, whereas ERRM and HRRM obtain much less replicas than RR for the high-heterogeneity application. In other words, ERRM and HRRM are better suitable for high-heterogeneity applications than for low-heterogeneity applications.

8 Conclusion

We developed enough and heuristic replication algorithms ERRM and HRRM to minimize the redundancy for a parallel application in heterogeneous service-oriented systems. The ERRM algorithm can enough minimize redundancy by presenting two-stage replications. To decrease the time complexity of the time-consuming ERRM algorithm, the HRRM algorithm was also presented to deal with large-scale parallel applications within a short time. The main advantage for HRRM is its capability to obtain lower sub-reliability requirements for most tasks compared with MaxRe and RR, such that HRRM can generate less redundancy than MaxRe and RR. Results of our experiments on real and random generaed parallel applications at different scales, parallelism degrees, and heterogeneity degrees validate that both ERRM and HRRM generate less redundancy than the state-of-the-art MaxRe and RR algorithms. Experiment results also show that the HRRM implements approximate optimal redundancy with a short computation time. We believe that the proposed algorithms can effectively facilitate a reliability-aware design for parallel applications in heterogeneous service-oriented systems.

Resource usage and shortest schedule length are also important concern in high-performance computing systems. In fact, minimum redundancy does not mean minimum resource usage and shortest schedule length for a parallel application on heterogeneous systems because the same task has different execution time values on different processors. In our future work, we will consider the resource usage and schedule length minimization in such environment.

Acknowledgments

The authors would like to express their gratitude to the anonymous reviewers for their constructive comments which have helped to improve the quality of the paper. This work was partially supported by the National Key Research and Development Plan of China under Grant Nos. 2016YFB0200405 and 2012AA01A301-01, the Natural Science Foundation of China under Grant Nos. 61672217, 61173036, 61432005, 61370095, 61300037, 61502405, 61300039, 61402170, and 61502162, the China Postdoctoral Science Foundation under Grant No. 2016M592422.

Supplement Material

The web page http://esnl.hnu.edu.cn/index.php/tsc/ publishes the experimental codes of the paper.

References

[1] Z. Cai, X. Li, and J. N. D. Gupta, "Heuristics for provisioning services to workflows in XaaS clouds," *IEEE Trans. Services Comput.*, vol. 9, no. 2, pp. 250–263, Mar./Apr. 2016.
[2] A. Zhou, S. Wang, B. Cheng, Z. Zheng, F. Yang, R. Chang, M. Lyu, and R. Buyya, "Cloud service reliability enhancement via virtual machine placement optimization," *IEEE Trans. Services Comput.*, vol. PP, no. 99, p. 1, Jan. 2016, doi: 10.1109/TSC.2016.2519898.
[3] Z. Fu, F. Huang, X. Sun, A. Vasilakos, and C.-N. Yang, "Enabling semantic search based on conceptual graphs over encrypted outsourced data," *IEEE Trans. Services Comput.*, vol. PP, no. 99, p. 1, Oct. 2016, doi: 10.1109/TSC.2016.2622697.
[4] Z. Xia, X. Wang, X. Sun, and Q. Wang, "A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 2, pp. 340–352, Feb. 2016.
[5] Y. Kong, M. Zhang, and D. Ye, "A belief propagation-based method for task allocation in open and dynamic cloud environments," *Knowl.-Based Syst.*, vol. 115, pp. 123–132, Jan. 2017.
[6] F. Zhangjie, S. Xingming, L. Qi, Z. Lu, and S. Jiangang, "Achieving efficient cloud search services: Multi-keyword ranked search over encrypted cloud data supporting parallel computing," *IEICE Trans. Commun.*, vol. 98, no. 1, pp. 190–200, Jan. 2015.
[7] Q. Liu, W. Cai, J. Shen, Z. Fu, X. Liu, and N. Linge, "A speculative approach to spatial-temporal efficiency with multi-objective optimization in a heterogeneous cloud environment," *Secur. Commun. Netw.*, vol. 9, no. 17, pp. 4002–4012, Nov. 2016.
[8] Z. Tang, L. Qi, Z. Cheng, K. Li, S. U. Khan, and K. Li, "An energy-efficient task scheduling algorithm in DVFS-enabled cloud environment," *J. Grid Comput.*, vol. 14, no. 1, pp. 55–74, Mar. 2016.
[9] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Aug. 2002.
[10] M. A. Khan, "Scheduling for heterogeneous systems using constrained critical paths," *Parallel Comput.*, vol. 38, no. 4, pp. 175–193, Apr. 2012.
[11] G. Xie, R. Li, and K. Li, "Heterogeneity-driven end-to-end synchronized scheduling for precedence constrained tasks and messages on networked embedded systems," *J. Parallel Distrib. Comput.*, vol. 83, pp. 1–12, May. 2015.

[12] G. Xie, X. Xiao, R. Li, and K. Li, "Schedule length minimization of parallel applications with energy consumption constraints using heuristics on heterogeneous distributed systems," *Concurrency Comput.-Parctice Experience*, pp. 1–10, Nov. 2016, doi: 10.1002/cpe.4024.

[13] J.-S. Leu, C.-F. Chen, and K.-C. Hsu, "Improving heterogeneous SOA-Based iot message stability by shortest processing time scheduling," *IEEE Trans. Services Comput.*, vol. 7, no. 4, pp. 575–585, Oct.-Dec. 2014.

[14] L. Zhao, Y. Ren, Y. Xiang, and K. Sakurai, "Fault-tolerant scheduling with dynamic number of replicas in heterogeneous systems," in *Proc. 12th IEEE Int. Conf. High Perform. Comput. Commun.*, 2010, pp. 434–441.

[15] L. Zhao, Y. Ren, and K. Sakurai, "Reliable workflow scheduling with less resource redundancy," *Parallel Comput.*, vol. 39, no. 10, pp. 567–585, Jul. 2013.

[16] Z. Zheng, T. C. Zhou, M. Lyu, and I. King, "Component ranking for fault-tolerant cloud applications," *IEEE Trans. Services Comput.*, vol. 5, no. 4, pp. 540–550, Oct.-Dec. 2012.

[17] W. Qiu, Z. Zheng, X. Wang, X. Yang, and M. R. Lyu, "Reliability-based design optimization for cloud migration," *IEEE Trans. Services Comput.*, vol. 7, no. 2, pp. 223–236, Apr.-Jun. 2014.

[18] M. Silic, G. Delac, and S. Srbljic, "Prediction of atomic web services reliability for QoS-Aware recommendation," *IEEE Trans. Services Comput.*, vol. 8, no. 3, pp. 425–438, May/Jun. 2015.

[19] A. Girault and H. Kalla, "A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate," *IEEE Trans. Dependable Secure Comput.*, vol. 6, no. 4, pp. 241–254, Oct.-Dec. 2009.

[20] A. Benoit, M. Hakem, and Y. Robert, "Fault tolerant scheduling of precedence task graphs on heterogeneous platforms," in *Proc. 22th IEEE Int. Parallel Distrib. Process.*, 2008, pp. 1–8.

[21] A. Benoit and M. Hakem, "Optimizing the latency of streaming applications under throughput and reliability constraints," in *Proc. 45th Int. Conf. Parallel Process.*, 2009, pp. 325–332.

[22] [Online]. Available: https://en.wikipedia.org/wiki/IEC_61508

[23] [Online]. Available: https://en.wikipedia.org/wiki/ISO_9000

[24] G. Xie, L. Liu, L. Yang, and R. Li, "Scheduling trade-off of dynamic multiple parallel workflows on heterogeneous distributed computing systems," *Concurrency Comput.-Parctice Experience*, vol. 29, no. 2, pp. 1–18, Jan. 2017, doi: 10.1002/cpe.3782.

[25] V. T'kindt and J.-C. Billaut, *Multicriteria Scheduling: Theory, Models and Algorithms*. Berlin, Germany: Springer, Mar. 2006.

[26] A. Doğan and F. Özgüner, "Biobjective scheduling algorithms for execution time–reliability trade-off in heterogeneous computing systems," *Comput. J.*, vol. 48, no. 3, pp. 300–314, Mar. 2005.

[27] M. Hakem and F. Butelle, "A bi-objective algorithm for scheduling parallel applications on heterogeneous systems subject to failures," in *Proc. RenPar2006*, 2006, pp. 25–35.

[28] J. J. Dongarra, E. Jeannot, E. Saule, and Z. Shi, "Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems," in *Proc. 19th ACM Int. Symp. Parallel Algorithms Architectures*, 2007, pp. 280–288.

[29] J. Broberg, S. Venugopal, and R. Buyya, "Market-oriented grids and utility computing: The state-of-the-art and future directions," *J. Grid Comput.*, vol. 6, no. 3, pp. 255–276, Sep. 2008.

[30] [Online]. Available: https://en.wikipedia.org/wiki/Service-level_agreement

[31] S. M. Shatz and J. P. Wang, "Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems," *IEEE Trans. Rel.*, vol. 38, no. 1, pp. 16–27, Apr. 1989.

[32] J. Mei, K. Li, X. Zhou, and K. Li, "Fault-tolerant dynamic rescheduling for heterogeneous computing systems," *J. Grid Comput.*, vol. 13, no. 4, pp. 507–525, Dec. 2015.

[33] X. Qin, H. Jiang, and D. R. Swanson, "An efficient fault-tolerant scheduling algorithm for real-time tasks with precedence constraints in heterogeneous systems," in *Proc. 31th Int. Conf. Parallel Process.*, 2002, pp. 360–368.

[34] X. Qin and H. Jiang, "A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems," *Parallel Comput.*, vol. 32, no. 5, pp. 331–356, Jun. 2006.

[35] Q. Zheng, B. Veeravalli, and C.-K. Tham, "On the design of fault-tolerant scheduling strategies using primary-backup approach for computational grids with low replication costs," *IEEE Trans. Comput.*, vol. 58, no. 3, pp. 380–393, Mar. 2009.

[36] A. Benoit, L.-C. Canon, E. Jeannot, and Y. Robert, "Reliability of task graph schedules with transient and fail-stop failures: Complexity and algorithms," *J. Scheduling*, vol. 15, no. 5, pp. 615–627, Oct. 2012.

[37] A. Verma and N. Bhardwaj, "A review on routing information protocol (RIP) and open shortest path first (OSPF) routing protocol," *Int. J. Future Generation Commun. Netw.*, vol. 9, no. 4, pp. 161–170, Apr. 2016.

[38] Q. Zheng and B. Veeravalli, "On the design of communication-aware fault-tolerant scheduling algorithms for precedence constrained tasks in grid computing systems with dedicated communication devices," *J. Parallel Distrib. Comput.*, vol. 69, no. 3, pp. 282–294, 2009.

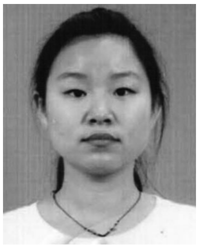
[39] L. Zhao, Y. Ren, and K. Sakurai, "A resource minimizing scheduling algorithm with ensuring the deadline and reliability in heterogeneous systems," in *Proc. 25th IEEE Int. Conf. Adv. Inf. Netw. Appl.*, 2011, pp. 275–282.

[40] [Online]. Available: https://sourceforge.net/projects/taskgraphgen/

Guoqi Xie received the PhD degree in computer science and engineering from Hunan University, China, in 2014. He was a postdoctoral researcher with Nagoya University, Japan, from 2014 to 2015. Since 2015, he is working as a postdoctoral researcher with Hunan University, China. He has received the best paper award from ISPA 2016. His major interests include embedded and real-time systems, parallel and distributed systems, software engineering and methodology. He is a member of the IEEE, the ACM, and the CCF.

Gang Zeng received the PhD degree in information science from Chiba University, in 2006. He is an associate professor in the Graduate School of Engineering, Nagoya University. From 2006 to 2010, he was a researcher, and then assistant professor in the Center for Embedded Computing Systems (NCES), Graduate School of Information Science, Nagoya University. His research interests mainly include power-aware computing and real-time embedded system design. He is a member of the IEEE and the IPSJ.

Yuekun Chen is currently working toward the PhD degree at Hunan University. Her research interests include services and cloud computing, fault-tolerance computing, and software engineering.



Yang Bai is currently working toward the PhD degree at Hunan Province, Hunan University. Her research interests include service computing, embedded systems, and cyber-physical systems.

Zhili Zhou is an assistant professor with Nanjing University of Information Science and Technology. His research interests include cloud computing, information forensics and security, pattern recognition.

Renfa Li is a professor of computer science and electronic engineering, and the dean of the College of Computer Science and Electronic Engineering, Hunan University, China. He is the director of the Key Laboratory for Embedded and Network Computing of Hunan Province, China. His major interests include computer architectures, embedded computing systems, cyber-physical systems, and Internet of things. He is a member of the council of CCF, a senior member of the IEEE, and a senior member of the ACM.

Keqin Li is a SUNY distinguished professor of computer science. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, Internet of things, and cyber-physical systems. He has published more than 460 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *IEEE Transactions on Cloud Computing*, the *IEEE Transactions on Services Computing*, and the *IEEE Transactions on Sustainable Computing*. He is a fellow of the IEEE.