JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 33, 81-99 (2017)
DOI: 10.6688/JISE.2017.33.1.6

A Reliability Analysis for Successful Execution of Parallel DAG Tasks

KE-KUN HU, GUO-SUN ZENG*, WEN-JUAN LIU AND WEI WANG
Department of Computer Science and Technology
Tongji University
Tongji Branch, National Engineering and Technology Center of High Performance Computer
Shanghai, 201804 P.R. China
E-mail: hookk@msn.com; {gszeng; wenjuanliu; wwang}@tongji.edu.cn

Large scale parallel computing system is becoming more and more failure-prone due to the increasing number of computational nodes. This results in serious reliability problems in parallel computing. To ensure successfully running of parallel tasks such as Meta tasks and DAG tasks, it is necessary to perform reliability analysis before scheduling parallel tasks. For Meta tasks, some key factors are discussed that affect and impede successful execution of a single task. Then, the reliability formula of Meta tasks is presented. For DAG tasks, hardware failures, software failures, network link failures and subtask execution order are all taken into account. We shall calculate not only the reliability of subtasks, but also the reliability of network communication. Then two reliability algorithms of DAG tasks are designed. Finally, some experiments are conducted. Experimental results show that our reliability analysis methods are more effective and comprehensive.

Keywords: parallel computing, meta tasks, DAG tasks, successful execution, reliability

1. INTRODUCTION

High Performance Computing (HPC) plays an important role in many fields, such as weather forecast, hydrocarbon exploration, biological modeling and simulation. These computing-intensive applications in turn have driven the scale of high performance computers from thousands to tens of thousands, or even hundreds of thousands of cores. For example, in 2016, the No. 1 supercomputer in the Top500 supercomputer list is Tianhe-2 consisting of 3, 123, 000 cores with an Rpeak of 54.902 PetaFlops [1]. This is an indisputable fact that scaling out the parallel computing system is an important way to improve its performance. However, as the scale of a parallel computing system grows, failures have become common. Los Alamos National Laboratory of America collects failure data of 22 high performance computers from 1996 to 2005, according to which the failure rate of single computational node is one in 512 hours [2]. But the runtime of many computing-intensive applications is weeks or even months. Therefore, it is difficult, if not impossible, for them to complete execution without failures. The frequent failures mean a waste of resources. Raicu *et al.* [3] point out that 20% or more of the computing capacity in a large scale high performance computing system is wasted due to failures.

Reliability is one of the most important performance features of a supercomputing system. It indicates the probability of the system to function without failure under stated

Received November 12, 2015; revised April 5 & June 21, 2016; accepted July 17, 2016.
Communicated by Jan-Jan Wu.
* Corresponding author.

conditions for a specified amount of time [4]. Reliability analysis usually adopts some methods such as statistics solutions to measure the probability. This information can provide a good reference regarding reliability optimization and maintainability of a system. In parallel computing, if some failure data and distributions are obtained, they can be used to help system architecture design, task scheduling, fault-tolerance algorithm design, and so on. Because of the serious reliability problems in large-scale parallel computing, like in cloud computing, it is necessary to perform the reliability analysis for parallel computing. However, it is a non-trivial work to analyze the reliability due to the varieties of faults coming from system hardware, software, and network communication links. Hence, the traditional reliability models [5-7] cannot be directly applied to analyzing the reliability of successfully running parallel tasks. In this paper, we focus on Meta tasks and DAG tasks, and analyze the reliabilities of successfully running them on a parallel computing system. The main contributions are as follows:

(1) We propose a reliability model for Meta tasks. Its reliability formula is also presented.
(2) We propose a comprehensive reliability model for DAG tasks. In addition to the hardware and software failures, our model also takes communication link failures into considerations due to the communication requirements among subtasks. Moreover, algorithms to evaluate the reliability of DAG TASKS are designed.
(3) We conduct some numerical experiments to identify the key factors that affect the reliability of successfully running of parallel tasks, and to compare our reliability models with others. The experimental results demonstrate that the proposed models are comprehensive and consistent with the statistics reported.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 introduces the parallel computing environments and some definitions. Meanwhile, some traditional reliability analysis methods are also presented. Section 4 and 5 analyze the reliabilities of Meta tasks and DAG tasks, respectively. Section 6 conducts extensive numerical experiments to identify the key factors that affect the reliability of parallel tasks under different situations. Section 7 concludes our work with some thoughts on future work.

2. RELATED WORK

The reliability of parallel computing is the probability that parallel tasks are successfully executed on parallel computing system. Low reliability means a waste of computing capacity due to failures and maintenances. To improve computing performance, many studies have been carried out. Pezoa *et al.* [8] proposed a rigorous probabilistic framework to analyze the hardware reliability of a distributed computing system in the presence of random server node failures. Xiong *et al.* [9] studied the parallel computing reliability by modeling the state transition process of a parallel computing system as a Markov chain. They calculated the reliability as the summation of probabilities of each state whose number of computational nodes in the operational state is larger than the maximum parallelism of parallel tasks. But References [8, 9] only considered the nodes

failure without taking the software into account. Xie *et al.* [10] proposed a reliability model for DAG tasks based on theory of probability. They assumed the failure events of each subtask could be modeled by non-homogeneous Poisson process while supposing the computational node to be perfect. However, just as the analysis of failure data collected from two large HPC sites suggested, hardware and software were two main reliability limiting factors [11]. Only considering either of them is not reasonable. Furthermore, References [9, 10] did not consider communication link failures. This is not appropriate for dependent tasks such as DAG tasks. Because the communication data among its subtasks may not be successfully transmitted through links in the presence of link failures. Shi *et al.* [12] defined the reliability of grid computing as the probability of successfully running of the tasks on different resources. They use reliability block diagram technique to evaluate the reliability. Dai *et al.* [13] defined the service reliability of grid computing system as the probability that all of the programs involved in the service are executed successfully. They derived the reliability using graphic theory. Guo *et al.* [14] studied the service reliability on a grid with star topology. They defined the reliability as the probability that the resource management system received all the outcomes of subtasks successfully. References [12-14] all assumed the failure rates of node and link obeyed the negative exponential distribution, but they did not consider software failures. As the scale of software grows, software failures become common. So, it is necessary to take software failures into consideration. Dai *et al.* [15] analyzed the cloud service reliability and defined it as the product of request stage reliability and execution stage reliability. In their reliability models proposed for each stage, they took the failures of hardware, software, and link into account. But they assumed no precedence constraints on the order of subtask execution. This assumption limits the application of their reliability model because parallel tasks often have dependencies among their subtasks.

In summary, parallel tasks reliability depends on both hardware and software. When there are data dependencies among subtasks, the communication link failures and precedence constraints on the order of subtask execution should also be considered. To our best knowledge, there are few literatures that take all these reliability limiting factors into account. Thus, in this paper, we present comprehensive reliability analysis for successfully running parallel tasks on a parallel computing system which take full considerations. Comparisons of related work are summarized in Table 1.

Table 1. Comparisons of related work.

reliability limiting factors / study objects and references		hardware	software	link	subtask execution order
system reliability	[8]	√	×	×	×
parallel computing reliability	[9]	√	×	×	√
	[10]	×	√	×	√
	[12]	√	×	√	√
	[13, 14]	√	×	√	×
	[15]	√	√	√	×
	Our work	√	√	√	√

√: taken into account ×: not taken into account.

3. RELIABILITY CONCEPTS IN A PARALLEL COMPUTING SYSTEM

3.1 Parallel Computing System

Parallel computing is the simultaneous use of multiple computational nodes to solve application problems [16]. Generally speaking, it has three steps. Firstly, an application problem is divided into small subtasks. Then these subtasks are assigned to different computational nodes. Finally, all these nodes cooperate and execute the assigned subtasks concurrently to speed up the problem solving process. A typical parallel computing system is shown in Fig. 1. From left to right, there are parallel tasks, task scheduler, parallel machine composed of computational nodes and high speed network, respectively. The task scheduler is responsible for mapping and scheduling parallel tasks to different computational nodes.

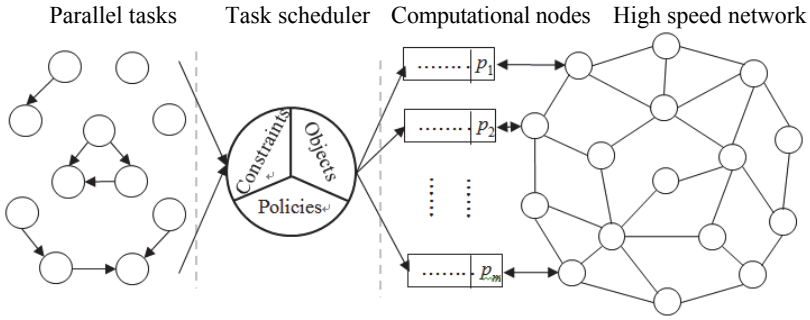


Fig. 1. A parallel computing system.

As to parallel tasks, they are just software programs running on different computational nodes to solve application problems. Parallel tasks can be classified into independent and dependent parallel tasks. The former refers to Meta tasks, the latter refers to DAG tasks. In this paper, when carrying out reliability analysis, we assume: (1) every task may fail due to some errors and the failures are statistically independent; and (2) each task has only two states: operational and failed.

A parallel machine is composed of computational nodes connected by high speed network. Each node can be a cheap personal computer, workstation, or cluster. The topology of the high speed network can be bus, ring, or hypercube, and so on. For convenience, we model a parallel machine as a quadruples $ARC=(P, L, S, B)$, where $P=\{p_i \mid i=1, 2, \ldots, m\}$ is the set of computational nodes and p_i denotes the ith node; $L=\{l_{uv} \mid u, v=1, 2, \ldots, m\}$ is the set of communication links, and l_{uv} indicates that there is a link between node p_u and p_v; $S=\{s_i \mid i=1, 2, \ldots, m\}$ denotes set of nodes' computational capability, and s_i corresponds to the computational capability of p_i; and $B=\{ b_{uv} \mid u, v=1, 2, \ldots, m\}$ is the set of link bandwidth, and b_{uv} denoting the bandwidth of l_{uv}. We make some assumptions for our target parallel machine: (1) it is composed of m nodes connected by high speed network with arbitrary topology; (2) both nodes and links may fail and have only two states: operational and failed; and (3) the failure of nodes and links are stochastic and statistically independent.

Tasks scheduler is actually a group of schedule policies and algorithms. Its responsibilities are to determine the execution sequence of multiple tasks and their corresponding execution time. There are many kinds of scheduler algorithms such as min-min, max-min. Of course, parallel tasks reliabilities vary with different scheduling algorithms. As scheduling algorithm is beyond the scope of this paper, we assume the tasks scheduler obeys the following principles: (1) it adopts a static centralized scheduling algorithm; (2) each task is only assigned to an idle node; and (3) each node can execute only one task at a time.

3.2 Definition of Reliability

Reliability has many different connotations. For an industrial product, it indicates the ability of the product to perform its expected function under specified conditions for a specified period of time. For a system, it indicates the probability that a system will satisfactorily perform its intended function. Similarly, we define the parallel computing reliability as follows:

Definition 1: *Parallel computing reliability* is the probability that parallel tasks will be executed successfully on parallel machines with a specified scheduling algorithm under stated conditions for a specified period of time.

Mathematically, the parallel computing reliability function $R(t)$ is the probability that the parallel tasks will run successfully without failure in the interval from time 0 to time t, $R(t)=P(T>t)$, $t\geq 0$, where T is a random variable representing the task failure time. Specifically, $R(t)=\exp\{-\int_0^t \lambda(x)dx\}$, where $\lambda(t)$ denotes the task failure rate function, or hazard function. Because hardware and software have a big influence on the parallel computing reliability, $\lambda(t)$ should take both of them into consideration.

3.3 Traditional Methods for Reliability Analysis

There are many different kinds of reliability analysis methods, such as statistical analysis, structure analysis, and simulations analysis [17].

(1) Statistical analysis: Suppose there are n system failures occurred during a time interval t. x_i and y_i (i=1, 2, ..., n) denotes the normal operation time and failure repair time of ith cycle, respectively, and the mean time between failures and the mean time to repair are $\frac{1}{n}\sum_{i=1}^{n} x_i$ and $\frac{1}{n}\sum_{i=1}^{n} y_i$, respectively. Thus, the system reliability, denoted by $r(t)$, can be calculated by $r(t) = (\frac{1}{n}\sum_{i=1}^{n} x_i)/(\frac{1}{n}\sum_{i=1}^{n} x_i + \frac{1}{n}\sum_{i=1}^{n} y_i)$.
(2) Structure analysis: Assume there is a serial-parallel connection system whose structure is shown in Fig. 2 and each component has a constant reliability p. Then the reliabilities of the nth group and the system are $r_p = 1-(1-p)^m$ and $r(t) = p^{n-1}(1-(1-p)^m)$, respectively.
(3) Simulation analysis: A method that approximates the system reliability by running a computer simulation program. A typical representative is Monte Carlo simulation. We omit its details because of the space limitation.

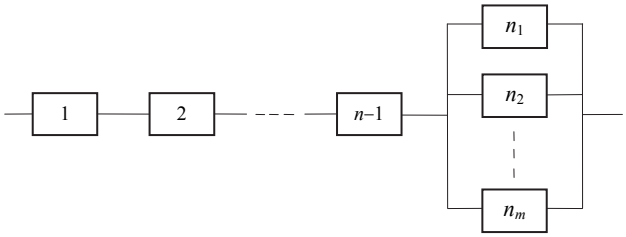


Fig. 2. Structure of a simple serial-parallel connection system.

4. RELIABILITY ANALYSIS FOR META TASKS

Usually there are many independent tasks in parallel computing, such as different services offered by cloud to user. The user may concern what the reliability of all the services being executed successfully is. This section performs the reliability analysis for this simple kind of parallel tasks, which establishes a foundation for that of dependent ones in the following section.

4.1 Definition of Meta Tasks and Basic Assumptions

Definition 2: *Meta tasks* is a kind of parallel tasks comprising a set of independent and indivisible single tasks. All these tasks can run on different computational nodes concurrently. It can be represented by a set $MT=\{t_j \mid j=1, 2, 3, \ldots, n\}$, where t_j denotes a task.

When Meta tasks are running on different computational nodes, one or more of them may fail due to faults. Because of the independency, a single task's failure will not affect any other single task. According to the theory of probability, the reliability that all Meta tasks are executed successfully, named Meta tasks parallel computing reliability in this paper, is equal to the product of each single task's reliability. Firstly, we analyze the simplest case: the reliability of a single task running on one computational node.

Whether or not a single task can run successfully, it depends on both hardware and software. Hardware refers to the physical components of a computational node such as CPU, memory, disk, while software refers to operation system, task program, and so on. Inspired by [18], we assume the hardware failure rate follows an exponential distribution. We use $r_i^h(t)$ to denote the hardware reliability of computational node p_i, then $r_i^h(t) = e^{-\alpha_i t}$, where α_i is the failure rate of p_i. Similarly, we also assume the software failure rate of a single task t_j follows an exponential distribution with parameter β_j. We denote the software reliability of t_j with $r_j^s(t)$, then $r_j^s(t) = e^{-\beta_j t}$. Furthermore, hardware and software failures are often independent with each other. Thus, the execution reliability of t_j on p_i, denoted by $r_{ij}(\tau_{ij})$, is equal to the product of hardware and software reliability:

$$r_{ij}(\tau_{ij}) = r_1^h(\tau_{ij}) \times r_j^2(\tau_{ij}) = e^{-(\alpha_i+\beta_j)\tau_{ij}} = e^{-\lambda_{ij}\tau_{ij}}, \quad (1)$$

where λ_{ij} and τ_{ij} represent the failure rate and execution time of t_j on p_i, respectively. Obviously, $\lambda_{ij} = \alpha_i + \beta_j$. It takes both hardware and software failures into account. Suppose the computational workload of single task t_j is w_j, then the execution time of t_j on p_i is $\tau_{ij}=w_j/s_i$, where s_i is the computational capability of node p_i. Thus, the reliability of a single task running on a single computational node can be calculated by Eq. (1).

4.2 Reliability of Meta Tasks

As discussed above, we analyze the reliability of a single task on one computational node. Based on that, we analyze the Meta tasks parallel computing reliability in this section. Suppose all m computational nodes of the parallel computing system are idle and the scheduler starts to schedule single tasks when t=0. According to the schedule rules (2) assumed in Section 3.1, the scheduler can schedule $min(n, m)$ single tasks a time. We only consider the case where $n \le m$ for the following two reasons: (1) m is often very large in cloud computing environment; (2) task scheduler can schedule the Meta tasks in batches when $n>m$. In the second case, the reliability is equal to the product of that of each batch. So we focus on the case of $n \le m$.

When n=1, m>1, there is only a single task t_j to be scheduled and executed. t_j fails if and only if it fails on all computational nodes. Thus, the whole parallel computing system can be considered as a connection system in parallel with m parts. We use $r^j_m(t)$ to denote the reliability of successfully running t_j on the system, and r^j_m, the corresponding execution time, where $r^j_m = \max\{\tau_{ij} = w_j/s_i \mid i = 1, 2, m\}$, then

$$r^j_m(\tau^i_m) = 1 - \prod_{i=1}^{m}\{1 - r_{ij}(\tau_{ij})\} = 1 - \prod_{i=1}^{m}(1 - e^{-\lambda_{i1}\tau_{i1}}). \tag{2}$$

When n>1, m>1, the successfully running of Meta tasks means all the single tasks are executed successfully. Without loss of generality, we assume they are completed in the order of $t_1, t_2, \ldots, t_n$. For single task t_1 ($t_2, \ldots, t_n$), the parallel computing system can be treated as a connection system in parallel with m (m–1, m–2, …, m–n+1) parts, respectively. We denote the Meta tasks parallel computing reliability with $r_{Meta}(t)$, then according to Eq. (1), we have

$$\begin{aligned} r_{Meta}(t) &= r^1_m(\tau^1_m)\cdot r^2_{m-1}(\tau^2_{m-1})\ldots r^n_{m-n+1}(\tau^n_{m-n+1}) \\ &= (1 - \prod_{i=1}^{m}(1 - e^{-\lambda_{i1}\tau_{i1}}))\cdot(1 - \prod_{i=2}^{m}(1 - e^{-\lambda_{i2}\tau_{i2}}))\ldots(1 - \prod_{i=n}^{m}(1 - e^{-\lambda_{in}\tau_{in}})), \end{aligned} \tag{3}$$

where τ^j_m denotes the finish time of t_j and t=max$\{\tau^j_m \mid j = 1, 2, \ldots, n\}$.

5. RELIABILITY ANALYSIS FOR DAG TASKS

Meta tasks is very simple parallel tasks. However, there are many other parallel tasks with complex dependencies among their subtasks such as Fork-Join [19], MapReduce [20], or Directed Acyclic Graph (DAG). Because DAG tasks is a generalization of Fork-Join and MapReduce tasks, we focus on the more general and expressive DAG tasks.

5.1 Definition of DAG Tasks and Basic Assumptions

Definition 3: *DAG tasks* consist of a set of subtasks with precedence constraints. We model a DAG tasks as a quadruple $DAG=(T, E, W, C)$, where (1) $T=\{t_j \mid j=1, 2, 3, \ldots, n\}$ is the set of subtasks and t_j denotes the jth subtask; (2) $E=\{e_{jk} \mid j, k=1, 2, 3, \ldots, n\}$ is the set of directed edges and e_{jk} characterize the data dependencies between subtasks t_j and t_k. t_j output its result to t_k and t_k takes the result as one of its inputs. t_j is said to be the immediate antecedent of t_k and t_k is referred as the immediate successor of t_j; (3) $W=\{w_j \mid j=1, 2,$

3, …, n} is the set of computational workload of subtasks and w_j denotes the computational workload of t_j; (4) $C=\{c_{jk} \mid j, k=1, 2, 3, \ldots, n\}$ is the set of data transfer volume, and c_{jk} denotes the data transfer volume between t_j and t_k.

Note that: here are some additional notation and terminology. We denote the immediate antecedent set and immediate successor set of t_j with $Pred(t_j)$ and $Succ(t_j)$, respectively. If $Pred(t_j)=\phi$, then t_j is called an entry subtask denoted by t_{entry}. If $Succ(t_j)=\phi$, then t_j is called an exit subtask denoted by t_{exit}.

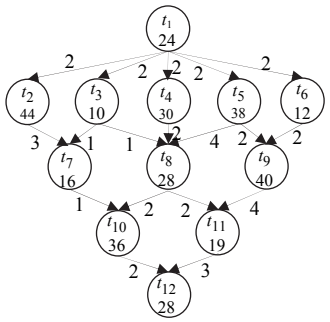


Fig. 3. An example of DAG tasks.

Fig. 3 gives an example of a DAG tasks. There are 12 subtask nodes with 18 edges connecting them. The numbers besides the edges represent the data transfer volume between subtasks. With zero in-degree, t_1 is the start point of the overall DAG tasks and spawns five subtasks. Every one of them then spawns more subtasks until reaching the exit subtask t_{12}, which indicates the finish of DAG tasks. During the execution, every subtask with nonzero in-degree starts to execute if and only if all its immediate antecedent subtasks execute successfully and send their results to it. Any subtask's failure will lead to the failure of the whole DAG tasks. Actually, there may be many entry subtasks and exit subtasks in a DAG tasks. However, it can be transformed to a DAG tasks with only one entry and exit subtask by adding one virtual entry subtask and one virtual exit subtask correspondingly. In this paper, we all refer to this type of DAG tasks unless stated otherwise.

Given the above DAG tasks model, we now analyze its parallel computing reliability on the parallel computing system. When $n>m$, like the scheduling of Meta tasks in batches, DAG tasks can be scheduled hierarchically, where the current layer subtasks take the output of upper layer subtasks as their input. Then the reliability of DAG tasks is equal to the product of that of each layer. So we focus on the case of $n \leq m$.

The successfully running of DAG tasks means all the subtasks are executed without failures and the network succeeds in transmitting all the communication data among subtasks. For each subtask, just as the case of single task of Meta tasks, its successful execution depends on both hardware and software. In addition, its successful execution implies all its immediate antecedent subtasks, if any, should also be executed successfully. This is different with Meta tasks. Furthermore, reliable network is another prerequisite for the successful data transmission among subtasks. Any communication link failure may fail the network in transmitting the data, which makes the immediate suc-

cessor subtasks unable to start due to lack of inputs. Thus, we can decompose the DAG tasks parallel computing reliability into two distinct parts: subtask execution reliability and network communication reliability, each of which can be calculated independently. Then, we obtain the DAG tasks parallel computing reliability by multiplying the former by the later.

5.2 Reliability of a Subtask Execution

Based on the discussion above, we firstly analyze how to calculate the execution reliability of any subtask. Because of the data dependencies, a subtask must receive all the results from its immediate antecedents (named the inputs receiving phase) and then begin to execute (named the execution phase). Upon finish, it should send the execution results to all its immediate successors (named the outputs sending phase). Any failure in any one of the three phases will definitely lead to failure of the subtask. As a subtask's reliability is affected by hardware and software, the successfully running of a subtask means both of them should be in the operational state all through these three phases. If we know the total time spent in these phases for each subtask, then we can calculate its reliability according to Eq. (1).

We now discuss how to calculate the total time. For convenience, this paper names the time spent in the inputs receiving and the outputs sending phases as communication time. Firstly, we analyze how to calculate it. Some basic assumptions are: (1) subtask t_j and t_k are assigned to computational node p_u and p_v, respectively; (2) the bandwidth of communication links between t_j and t_k is b_{uv} when $t_k \in Pred(t_j)$ and b_{vu} when $t_j \in Pred(t_k)$; and (3) the data size of inputs and outputs of subtask t_j are $d_{in}(t_j)$ and $d_{out}(t_j)$, respectively. We denote the bandwidth used to transmit all these data by bw. Then the communication time of t_j, denoted by $\tau_{comm}(t_j)$, is computed by

$$\tau_{comm}(t_j) = \frac{d_{in}(t_j) + d_{out}(t_j)}{bw} = \sum_{t_k \in Pred(t_j)} \frac{c_{kj}}{b_{uv}} + \sum_{t_k \in Succ(t_j)} \frac{c_{jk}}{b_{vu}}, \tag{4}$$

where $d_{in}(t_{entry})=d_{out}(t_{exit})=0$, because t_{entry} do not have inputs receiving stage and t_{exit} do not have outputs sending stage.

The execution time of t_j on p_i, denoted by $\tau_{comp}(t_j)$, is equal to w_j/s_i. We denote the total time as τ_{ij}, then

$$\tau_{ij} = \tau_{comp}(t_j) + \tau_{comm}(t_j) = \frac{w_j}{v_i} + \sum_{t_k \in Pred(t_j)} \frac{c_{kj}}{b_{uv}} + \sum_{t_k \in Succ(t_j)} \frac{c_{jk}}{b_{vu}}. \tag{5}$$

So the execution reliability of t_j on p_i is equal to $e^{-\lambda_{ij}\tau_{ij}}$ according to Eq. (1). However, as discussed above, we should also take the execution reliabilities of all t_j's immediate antecedent subtasks into account. We denote the computational nodes to which the scheduler assigns subtask $t_k \in Pred(t_j)$ by p_x, where $p_x \in P$ and $p_x \neq p_i$. Then the execution reliability of t_j, denoted by $r_{ij}(\tau_{ij})$, can be computed by

$$r_{ij}(\tau_{ij}) = \prod_{t_k \in Pred(t_j), p_x \in P} r_{xk}(\tau_{xk}) \times e^{-\lambda_{ij}\tau_{ij}}. \tag{6}$$

Obviously, any subtask's execution reliability can be calculated by Eq. (6) recursively

until reaching the exit subtask. The pseudo code of the algorithm to calculate any subtask's execution reliability is given as follows:

Algorithm 1: *Subtask_Execution_Reliability_Algorithm*, *SERA*

Input: DAG=(T, E, C, W), ARC=(P, L, S, B), λ_{ij}; //Follow the assumptions in the previous section

Output: $r_{ij}(\tau_{ij})$ ($1 \le i \le m$, $1 \le j \le n$), r_{Exit}, t_{total}; //t_{total} is the total execution time of DAG tasks

```
SERA ( )
{   P←{p1, p2, …, pm}; //Initialize set P that denotes the computational nodes in idle state
    L←{tj | in-degree(tj)==0, 1≤ j ≤ n }; //Add subtask with zero in-degree to the task ready queue L
    τs(tj) ← 0, τe(tj) ← 0, j=1, 2, 3, …, n;
    τidle(pi) ← 0, i=1, 2, 3, …, m;
    ttotal←0;
    ε←ϕ;
    do until L is empty
    {   for each idle processor node pi (1≤ i ≤ m) //Assign idle computational nodes for subtasks in L
        {   tj ← scheduling(pi, L); //Implement task scheduling according to schedule policies
            if (tj > 0)
            {   L←L−{ tj }, ε←ε+{tj};
                τs(tj) ← max( τs (tj), τidle(pi) );
            }
        }
        for each tj ∈ ε //Calculate the execution time and reliability of tj
        {   ε ← ε − {tj};
            τe(tj)← τs (tj)+ τcomp(tj)+ τcomm(tj); //Calculate the ending time of tj
            τidle(pi)← τe(tj);
            τij= τe(tj)− τs(tj); //Calculate the execution time of tj
            if (tj=tentry) //If tj is the entry subtask
                rij( τij)←e^(-λij τij);
            else
            {   Pred(tj)←find_immediate_antecedent (tj, DAG);
                for each tk ∈ Pred(tj) //Retrieve the calculated subtask execution relability
                    rxk( τxk)←rxk( τxk)←get_finished_reliability (tk, tj);
                rij(τij) = ∏tk∈Pred(tj),ps∈P rxk(τxk) × e^(−λij τij); //current subtask execution reliability
            }
        }
        for each tk ∈ Succ(tj) //Update some parameters regarding tk
        {   τs(tk) ← max( τs(tk), τidle(pi));
```

```
            indegree(t_k) ← indegree(t_k)−1;        //Update the in-degree of t_k
            if (indegree(t_k)==0)  L←L+{t_k};       //If the in-degree of t_k decreases to
                                                      zero, then adds it to L
        }
    }
    t_total← τ_e(t_exit);              //The ending time of t_exit is equal to the total execution
                                         time of DAG tasks
    return {r_ij(τ_ij)| (1≤ i ≤ m, 1≤ j ≤ n)}, t_total;     //Output the results
}
```

Algorithm 1 traverses the DAG from the entry subtask in breadth-first manner. Meanwhile, it maintains a task ready queue and assigns each of them to an idle computational node. At the same time, it records the starting and ending time for each subtask. Based on these information, it calculates the execution reliability of every subtask in the queue recursively until reaching the exit subtask. Obviously, the successfully running of the exit subtask means that all subtasks have been executed successfully. Thus, the execution reliability of t_{exit}, denoted by $r_{Exit}(\tau_{xn})$, represents the execution reliability of the whole DAG tasks.

5.3 Reliability of Network Communication

The reliable communication network is a guarantee of the successful data transmission among subtasks. Thus, the reliability of network communication should also be taken into consideration. The network is reliable if and only if it satisfies the communication requirements between any pair of subtasks. This requires that there is at least one communication path between any pair of subtasks. However, communication links may encounter failures due to radiation effect, wear-out, or aging problems. This may result in the failures of data transmission. To calculate the reliability of network communication, we assume: (1) every communication link may experience failures, and they are independent with each other. Every communication link has only two states: operational state with the probability ρ and failed state with the probability $1-\rho$; (2) The failure rate of communication link follows an exponential distribution. We denote the reliability of successful data transmission of l_{uv} with $r_{uv}^{l}(\tau_{uv}^{l})$, then $r_{vu}^{l}(\tau_{uv}^{l}) = e^{-\gamma_{uv}\tau_{uv}^{l}}$, where γ_{uv} represents the failure rate of l_{uv}. And τ_{uv}^{l} represents the transmission time for data d_{uv} from computational node p_u to p_v, which can be calculated by $\tau_{uv}^{l} = \frac{d_{uv}}{b_{uv}}$; (3) Communication data can only be transmitted on the links whose two terminals both connect to computational node assigned subtasks. We call all these links along with computational nodes connected to them a subnet; (4) When the number of computational nodes m is larger than that of subtasks n, scheduler partitions the communication network into $\lfloor \frac{m}{n} \rfloor$ disjoint subnets.

If a subnet can satisfy the communication requirements of DAG tasks, we think that it is in the operational state. If there are more than one subnet in the operational state, the network communication reliability is determined by all of them. In this case, we can compute the average value of all subnets' communication reliabilities as the network communication reliability. Particularly, Algorithm 1 finds out a subnet that not only can successfully execute subtasks but also satisfy their communication demands. Suppose the

subnet is denoted by $ARC_q(P_q, L_q)$, where P_q and L_q represent the computational nodes set to which the scheduler assigns subtasks and communication links set that connect those computational nodes, respectively. Because $ARC_q(P_q, L_q)$ satisfies the communication requirements of DA in the operational state G tasks, the links in the operational state must form a spanning tree at least. Of course, there may be multiple spanning subgraphs in the operational state. We suppose the set of spanning subgraphs of $ARC_q(P_q, L_q)$ in the operational state is denoted by Ω_q and the set of links in the operational state is denoted by L_q', then the communication reliability of $ARC_q(P_q, L_q)$, denoted by $r^q_{Comm}(t)$, can be calculated as follows:

$$r^q_{Comm}(t_q) = \sum_{\Omega_q} \left(\prod_{l_{uv} \in L'_q} r^l_{uv}(\tau^l_{uv})\right) \times \prod_{l_{gh} \in (L_q \setminus L'_q)} (1 - r^l_{gh}(\tau^l_{gh})), \tag{7}$$

where t_q represents the execution time of DAG tasks. Its value equals to t_{total} computed by Algorithm 1.

We have discussed the single subnet communication reliability. In fact, when the number of computational nodes m is larger than that of subtasks n, the communication network is partitioned into $\lfloor \frac{m}{n} \rfloor$ disjoint subnets according to the assumption (4) in this section. Then the network communication reliability, denoted by $r_{Comm}(t_q)$, can be calculated by

$$r_{Comm}(t) = \frac{\sum_{ARC_q \in ARC[\lfloor \frac{m}{n} \rfloor]} r^q_{Com}(t_q)}{\lfloor \frac{m}{n} \rfloor}, \tag{8}$$

where $t=max\{t_q\}$. Based on discussion above, we design another algorithm, *Network_Communication_Reliability_Algorithm*, to calculate the network communication reliability. The following is its description in pseudo code.

Algorithm 2: *Network_Communication_Reliability_Algorithm*, *NCRA*

Input: $DAG=(T, E, C, W)$, $ARC=(P, L, S, B)$, γ_{uv}; //Follow the assumptions in the previous section

Output: $r_{Comm}(t)$; //The reliability of network communication

```
NCRA ( )
{  P←{p1, p2, …, pm};
   L←{ t1, t2, …, tn };                          //Initialize the task ready queue L
   r^q_Comm(tq)=0 (q=1, 2, 3, …, ⌊m/n⌋);
   ARC[⌊m/n⌋]←{ϕ, ϕ, …, ϕ};                      //Initialize the set of subnets
   Ω[⌊m/n⌋]←{ϕ, ϕ, …, ϕ};      //Initialize set of spanning subgraphs of each subnet in the
                                  operational state
   ARC[⌊m/n⌋]←subnet_decision_making(DAG, ARC); //Partition the system into subnets
   if (m<n)                                      //No subtasks to be scheduled
        r_Comm(t)←0;
   else if ⌊m/n⌋ ≥ 1                             //There are multiple subnets that may sat-
                                                   isfy communication reqirements
   {    for each subnet ARC_q ∈ ARC[⌊m/n⌋]        //Calculate communication reliability for
                                                   each subnet
```

{ if (is_connected(ARC_q)<0);
$r^q_{Comm}(t) \leftarrow 0$;
else
{ $\Omega_q \leftarrow$ generate_spanning_graph_in_operational_state (ARC_q);
$r^q_{Comm}(t_q) = \sum_{\Omega_q} (\prod_{l_{uv} \in L'_q} r^l_{uv}(\tau^l_{uv})) \times \prod_{l_{gh} \in (L_q \setminus L'_q)} (1 - r^l_{gh}(\tau^l_{gh}))$;
}
}
$r_{Comm}(t) \leftarrow \frac{\sum_{N_q \in N[\lfloor \frac{m}{n} \rfloor]} r^q_{Com}(t_q)}{\lfloor \frac{m}{n} \rfloor}$; //Calculate network communication reliability
}
return $r_{Comm}(t)$; //Output the network communication reliability
}

The following example is designed to ease the understanding of Algorithm 2.

Example 1: There is a DAG tasks with 4 subtasks and a parallel computing system with 6 computational nodes as are shown in Fig. 4. At the beginning, all computational nodes are idle and the scheduler schedules the DAG tasks to the system. The scheduler partitions the system into $\lfloor \frac{6}{4} \rfloor = 1$ subnet and assigns the DAG tasks to it. Suppose the subnet, denoted by $N_{1,}$ is composed of p_1, p_2, p_3, p_4 and the links connected among them, as indicated by dashed circle in Fig. 4. Obviously, N_1 has three spanning trees, represented by L^1_1={$l_{4,8}$, $l_{7,8}$, $l_{8,10}$}, L^2_1={$l_{4,7}$, $l_{7,8}$, $l_{8,10}$}, L^3_1={$l_{4,7}$, $l_{4,8}$, $l_{8,10}$}, respectively. In addition, L_1={ $l_{4,7}$, $l_{4,8}$, $l_{7,8}$, $l_{8,10}$} also satisfies the communication demand of the DAG tasks, so L_1 is in the operational state. Then Ω_1, the set of subnets of N_1 in the operational state, is {L^1_1, L^2_1, L^3_1, L_1}. To calculate N_1's communication reliability, we assume that all links have same bandwidth and failure rate, and b=1, γ_{uv} = 0.02 for simplicity. Because it is difficult to calculate data transfer volume on each links accurately, we take the average value as an approximation. For each link of L^1_1, the average data transfer volume is $\overline{c} = \frac{c}{|L^1_1|} = \frac{\sum_i^4 \sum_j^4 c_{ij}}{|L^1_1|}$ $= \frac{3 \times 2 + 3 + 1}{3} \approx 3.3$, and then the average communication time is $\overline{\tau}^L_{uv} = \frac{\overline{c}}{b} = \frac{3.3}{1} = 3.3$. Similarly, we can calculate the average communication time for each link of L^2_1, L^3_1, and L_1. Thus, we have $r^1_{Comm}(t_1)$ = $3 \times 0.9361^3 \times (1-0.9361) + 0.9512^4$= 0.9759 according to Eq. (7). Because there is only one subnet, the network communication reliability equals to $r^1_{Comm}(t_1)$.

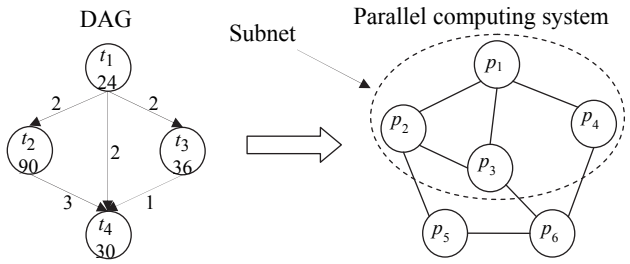


Fig. 4. A simple example illustrating how to calculate reliability of network communication.

5.4 Reliability of the Whole DAG Tasks

Till now, we have calculated the execution reliability of any subtask and reliability of network communication. Because the reliability of the whole DAG tasks, denoted by $r_{DAG}(t)$, is determined by both of them, then

$$r_{DAG}(t) = r_{Exit}(t) \times r_{Comm}(t), \tag{9}$$

where t represents the overall execution time of DAG tasks.

6. EXPERIMENTS AND APPLICATIONS

We have proposed reliability analysis methods for Meta tasks and DAG tasks in Sections 4 and 5, respectively. In order to verify our methods, we first design two experiments to investigate the influence of different hardware and software failure rates, and the number of single tasks on the Meta tasks parallel computing reliability. Then we design another three experiments for DAG tasks. The first one is to investigate the effect of different link failure rates on network communication reliability, and the second one is to study the effect of different failure rates of hardware, software, and link on DAG tasks reliability. The last one is designed to compare our reliability methods with others. Assume the parallel computing system we use is same as in Fig. 4.

6.1 Experiments for Meta Tasks

Experiment 1: This experiment is designed to study how the Meta tasks parallel computing reliability change with different hardware and software failure rates. Suppose four tasks t_1, t_2, t_3, and t_4 with different software failure rates, are assigned to computational nodes p_1, p_2, p_3, p_4, respectively. And p_5, p_6 have no tasks to be executed. The parameters of Meta tasks parallel computing are listed in Table 2. α_i denotes hardware failure rate of computational node p_i. β_j denotes software failure rate of task t_j, and λ_{ij} denotes the failure rate of t_j executing on p_i.

Table 2. Parameters of Meta tasks parallel computing.

	$p_1(t_1)$	$p_2(t_2)$	$p_3(t_3)$	$p_4(t_4)$	$p_5(t_5)$	p_6 (–)
α_i (/hr)	0.002μ	0.002μ	0.002μ	0.002μ	0.002μ	0.002μ
β_j(/hr)	0.009μ	0.015μ	0.020μ	0.010μ	–	–
λ_{ij}(/hr)	0.011μ	0.017μ	0.022μ	0.012μ	–	–

In this example we suppose all single tasks have an identical execution time, *i.e.*, $\tau_j \approx \tau$ (j=1, 2, 3, 4). Then the Meta tasks parallel computing reliability with respect to execution time are calculated according to Eq. (3). The results are shown in Fig. 5. It shows that the reliability decreases quickly with the increase of execution time. And the higher the failure rate is, the faster the reliability decreases. For instance, in the early stage of execution when t<10, the reliabilities with all three different failure rates are close to 1. However, they all approach to 0 when t = 350 hours. This means that it is almost impos-

sible for the Meta tasks to run so long time without failures. The possible reasons include computation errors of CPU resulting from increasing temperature, memory leaks, hard disk failures, and exposure of bugs in the code of tasks, and so on.

Experiment 2: We design this experiment to study how the Meta tasks parallel computing reliability change with different number of single tasks. In order to highlight the effects of number of single tasks on the reliability, we assume the failure rate of each single task when executing on a computational node is equal to 0.02. Then we calculate the Meta tasks parallel computing reliability according to Eq. (2). The results are shown in Fig. 6. It shows that the reliability decrease with the increase of execution time, no matter how many single tasks there are. What's more, the larger the number of single tasks is, the faster the reliability decreases. This is because when the number of single tasks is smaller than that of computational nodes, the rest of nodes serve as standby. When a single task encounters failures on one computational node, it can resume its execution on another node in idle state. This will increase its reliability.

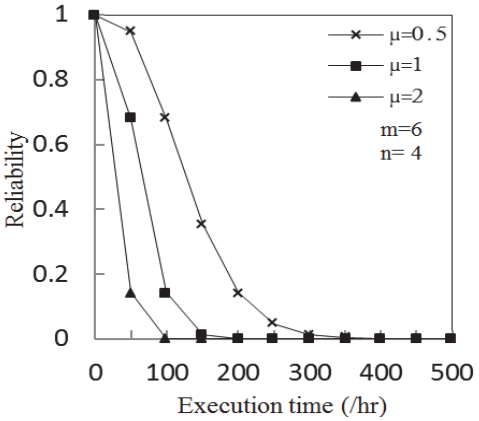


Fig. 5. Meta tasks reliability with different failure rates.

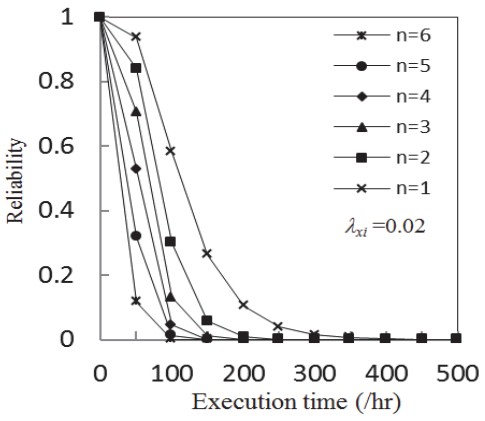


Fig. 6. Meta tasks reliability with different number of single tasks.

6.2 Experiments for DAG Tasks

Experiment 3: This experiment is designed to study how the network communication reliability change with different failure rates of communication links. We take the DAG tasks in Fig. 4 as our test object. They are scheduled to subnet $N_1(P_1, L_1)$, where $P_1 = \{p_1, p_2, p_3, p_4\}$ and $L_1 = \{l_{12}, l_{13}, l_{14}, l_{23}\}$. Suppose the failure rates of l_{12}, l_{13}, l_{14}, and l_{23} to be 0.012μ, 0.018μ, 0.020μ, 0.014μ, respectively. Then the network communication reliabilities with different link failure rates are calculated according to Eqs. (7) and (8). The results are shown in Fig. 7. Just as the result shown in Fig. 5, the reliabilities with three different link failure rates decrease quickly with the increase of execution time in Fig. 7. And the one with higher failure rate drops faster than others with lower failure rate. When t=250 hours, the reliabilities with all three different failure rates are close to 0. It is not acceptable for large-scale parallel applications whose execution time are weeks or

even months. Thus, the reliability of communication links has a big influence on the successfully running of DAG tasks and so it cannot be neglected.

Experiment 4: We design this experiment to study how the DAG tasks parallel computing reliability change with different hardware and software failure rates. The failure rates of hardware are assumed to be same with Experiment 1. DAG tasks are assumed to be same as in Fig. 4. And its subtasks have identical failure rates with the corresponding tasks in Experiment 1. For simplicity, we assume the ratio of each subtask's execution time towards the total execution time of DAG tasks are 1/5, 1/2, 1/2 and 1/5 for t_1, t_2, t_3, and t_4, respectively. The executions of t_2 and t_3 are completely overlapped. The ratio of communication time to total execution time of DAG tasks is assumed to be 1/10. Then the DAG tasks parallel computing reliability with respect to total execution time can be calculated according to Eqs. (4)-(9). Their results are shown in Fig. 8 in solid line. Similar to the result shown in Fig. 5, the DAG tasks parallel computing reliability calculated by our method decreases quickly with the increase of total execution time. However, the reliability drops much faster than that in Fig. 5. For example, the reliability of the former is 0.679525 while the latter is only 0.239766 in the case of μ=1, t=50. This is because the reliability of communication links has a big influence on the successfully running of DAG tasks.

Experiment 5: This experiment is designed to compare our reliability analysis method for DAG tasks reliability with others. Reference [10] is the most relevant one to our work. Thus, we use their reliability analysis method to calculate the reliabilities with the same parameters and configurations as in Experiment 4. Their results are shown in dotted line in Fig. 8. It can be noted that the reliabilities calculated by [10] is much higher than ours in Fig. 8. For example, the reliability is 0.4798 while it is almost 0 in our method in the case of μ=1, t=500. It implies that the probability that the DAG tasks can keep running as long as 500 hours without failures is nearly 0.5. It is not consistent with the statistics in [2]. Similar results can be obtained by the reliability analysis method provided by [9]. This is because both of them did not consider the communication link failures.

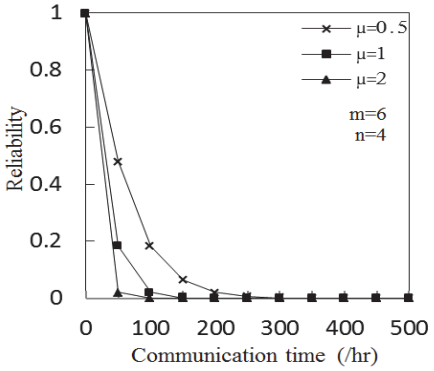


Fig. 7. Network communication reliability with different failure rates.

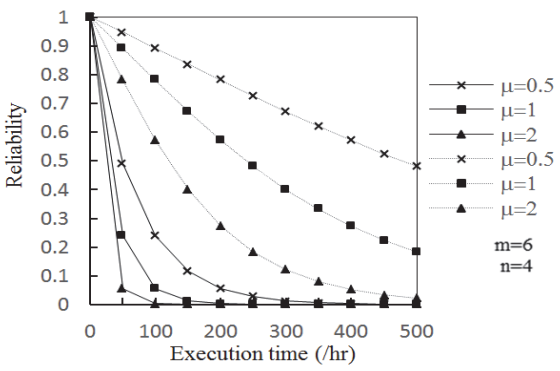


Fig. 8. DAG tasks reliability with different failure rates (solid line for our method, dotted line for [10]).

According to the above experimental results, we can know that our method is more consistent with actual statistics due to a comprehensive consideration of reliability limiting factors including the hardware failures, software failures, link failures, and constraints on the subtask execution order.

6.3 Possible Applications

Our reliability analysis methods may be useful in at least three fields: system architecture design, task scheduling, and fault-tolerance algorithm design. For system architecture design, designers can use our methods to identify key components (hardware, software, and communication link) that influence reliability greatly, and then deploy more backups for the less reliable ones to enhance the reliability. For task scheduling, scheduler can compute the reliabilities for different task assignments based on our methods, and select the one that has largest reliability and matches the task QoS requirements. If there are no assignments that satisfy the QoS requirements, then system maintainers can replace the unreliable components with more reliable ones or increase the number of backups. With respect to fault-tolerance algorithm design, programmers can compute the reliabilities with different parameters of fault-tolerance algorithm such as number of computational nodes, task replications. Based on some economic model, they can design an optimal fault-tolerance algorithm.

7. CONCLUSIONS

In this paper we have performed reliability analysis for Meta tasks and DAG tasks. For Meta tasks, the reliability of a single task on one computational node is firstly analyzed, then the reliability formula based on the theory of probability is presented. For DAG tasks, we propose a reliability model that takes a comprehensive consideration on the reliability limiting factors including computational nodes, software, communication links and subtask execution order. In this model, we calculate both the reliability of subtask execution and that of network communication. Furthermore, we design two algorithms, *SERA* and *NCRA*, to calculate the corresponding reliabilities. Finally, we conduct some numerical experiments to validate our reliability models. Our works have a guiding significance for reliability optimization in the field of high performance computing systems. On the other hand, this paper does not consider the fault tolerance mechanism and the interactions between hardware and software failures. And that will be our future work.

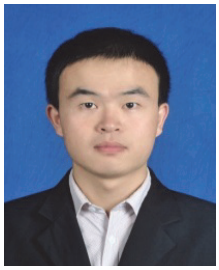
ACKNOWLEDGEMENTS

This work was supported by the National High-Tech Research and Development Plan of China under grant No. 2009AA012201; the Program of Shanghai Subject Chief Scientist under grant No. 10XD1404400; the Huawei Innovation Research Program under grant No. IRP-2013-12-03; the State Key Laboratory of High-end Server and Storage Technology under grant No. 2014HSSA10.

REFERENCES

1. The TOP500 supercomputer list, https://www.top500.org/system/177999.
2. The computer failure data repository, https://www.usenix.org/cfdr-data.
3. I. Raicu, I. T. Foster, and P. Beckman, "Making a case for distributed file systems at exascale," in *Proceedings of the 3rd ACM International Workshop on Large-Scale System and Application Performance*, 2011, pp. 11-18.
4. X. Yang, Z. Wang, J. Xue, *et al.*, "The reliability wall for exascale supercomputing," *IEEE Transactions on Computers*, Vol. 61, 2012, pp. 767-779.
5. W. D. Vanel, M. Schuld, R. Wijgers, *et al.*, "Software reliability and its interaction with hardware reliability," in *Proceedings of the 15th International Conference on Thermal*, *Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems*, 2014, pp. 1-8.
6. S. Thirumurugan and D. R. P. Williams, "Analysis of testing and operational software reliability in SRGM based on NHPP," *International Journal of Computer and Information Engineering*, Vol. 1, 2007, pp. 284-289.
7. J. Silva, T. Gomes, D. Tipper, *et al.*, "An effective algorithm for computing all-terminal reliability bounds," *Networks*, Vol. 66, 2015, pp. 282-295.
8. J. E. Pezoa, S. Dhakal, and M. M. Hayat, "Maximizing service reliability in distributed computing systems with random node failures: Theory and implementation," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 21, 2010, 1531-1544.
9. H. Xiong, G. Zeng, W. Wang, *et al.*, "Upper limit analysis of scalable parallel computing on the premise of reliability requirement," *IETE Technical Review*, 2016, pp. 1-11.
10. G. Q. Xie, R. F. Li, L. Liu, and F. Yang, "DAG reliability model and fault-tolerant algorithm for heterogeneous distributed systems," *Chinese Journal of Computers*, Vol. 36, 2013, pp. 2019-2032.
11. B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, Vol. 7, 2010, pp. 337-350.
12. X. Shi, H. Jin, W. Qiang, *et al.*, "Reliability analysis for grid computing," *Grid and Cooperative Computing*, Springer, Berlin Heidelberg, 2004, pp. 787-790.
13. Y. S. Dai, M. Xie, and K. L. Poh, "Reliability of grid service systems," *Computers and Industrial Engineering*, Vol. 50, 2006, pp. 130-147.
14. S. Guo, H. Z. Huang, Z. Wang, *et al.*, "Grid service reliability modeling and optimal task scheduling considering fault recovery," *IEEE Transactions on Reliability*, Vol. 60, 2011, pp. 263-274.
15. Y. S. Dai, B. Yang, J. Dongarra, and G. Zhang, "Cloud service reliability: Modeling and analysis," in *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, 2009, pp. 1-17.
16. L. Zhang, X. B. Chi, Z. Y. Mo, and N. Li, *Introduction to Parallel Computing*, Tsinghua University Press, Peking, 2006.
17. H. L. Zhang, "Research on key technologies of reliability analysis of dynamic systems," Graduate School, National University of Defense Technology, Changsha, 2011.
18. Q. Kang and H. He, "A novel discrete particle swarm optimization algorithm for

Meta tasks assignment in heterogeneous computing systems," *Microprocessors and Microsystems*, Vol. 35, 2011, pp. 10-17.
19. L. Yuan, P. Jia, and Y. Yang, "Scheduling of fork-join tasks on multi-core processors to avoid communication conflict," TENCON 2015-2015 IEEE Region 10 Conference, 2015, pp. 1-6.
20. J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, Vol. 51, 2008, pp. 107-113.



Ke-Kun Hu (胡克坤) received the MS degree in Computer Science from Shandong University of Science and Technology in 2014. He is currently working toward the Ph.D. degree in Computer Science at Tongji University. His research interests include parallel computing, system reliability and fault tolerance.

Guo-Sun Zeng (曾国荪) received the BS, MS, and Ph.D. degrees in Computer Software and Application all from the Department of Computer Science and Engineering, Shanghai Jiao Tong University. He is currently working at Tongji University as a Full Professor, and as a Supervisor of Ph.D. candidates in Computer Software and Theory. His research interests include green computing, parallel computing and information security. He is a senior member of the IEEE.

Wen-Juan Liu (刘文娟) was born in 1989 and received her M.S. in Wuhan University in 2013. Now she is a Ph.D. candidate in Department of Computer Science and Technology, Tongji University. Her research interests are focused on parallel distributed computing and system scaling.

Wei Wang (王伟) received his Ph.D. in Computer Software and Theory from the Department of Computer Science and Technology, Tongji University. He is an Associate Professor in Department of Computer Science and Technology at Tongji University, P.R. China. His research interests include parallel and distributed computing and information security. In 2007, he was granted IBM Ph.D. Fellowship.