Memetic Computing (2019) 11:357–370
https://doi.org/10.1007/s12293-019-00284-3

REGULAR RESEARCH PAPER

# An intelligent scheduling algorithm for complex manufacturing system simulation with frequent synchronizations in a cloud environment

Feng Yao[1] · Yiping Yao[1] · Lining Xing[1,2] · Huangke Chen[1] · Zhongwei Lin[3] · Tianlin Li[1]

Received: 27 October 2018 / Accepted: 20 March 2019 / Published online: 29 March 2019

**Abstract**
For cloud-based, large-scale complex manufacturing system simulation (CMSS), allocating appropriate service instances (virtual machines or nodes) is a promising way to improve execution efficiency. However, the complex interactions among and frequent aperiodic synchronizations of the entities of a CMSS make it challenging to estimate the influence of service instances' computing power and network latency on the execution efficiency. This hinders the appropriate allocation of service instances for CMSS. To solve this problem, we construct a performance estimation model (PEM) using the executed events and synchronization algorithms to evaluate the running time of CMSS on different service instance combinations. Further, an intelligent scheduling algorithm that introduces PEM as fitness function is proposed to search for a near-optimal allocation scheme of CMSS service instances. To be specific, the PEM-based optimization algorithm (PEMOA) incorporates simulated annealing into the mutation phase of a genetic algorithm to strengthen its local searching ability. A series of experiments were performed on a computer cluster to compare the proposed PEMOA with two representative algorithms: an adapted first-come-first-service-based and the max-min-based allocation algorithms. The experimental results demonstrate that the PEMOA can reduce the running time by more than 7%. In particular, the improvement of PEMOA increases when the manufacturing system simulation is communication-intensive or spans a small number of service instance combinations.

**Keywords** Frequent synchronizations · Intelligent manufacturing · Manufacturing system · Performance estimation · Parallel and distributed simulation · Resource allocation

## 1 Introduction

In a manufacturing system, the life-time of a product often involves various stages, such as design, simulation, manufacturing, transportation, and management [1]. Simulation has an important role in this process and can benefit the design and optimization of a manufacturing system. Thus, it has attracted much attention from researchers [2]. Negahban [3] demonstrated that simulation is helpful during short- and long-term production planning for manufacturing decision-makers. For a manufacturing system, reducing the time to market of products is a constant challenge [4]. A key strategy is to accelerate the simulation execution, and hence more efficient simulation execution is constantly pursued. In the literature [5], discrete event simulation was proposed as a feature of next-generation simulations in manufacturing systems. Parallel and distributed simulation (PADS), an important branch of discrete event simulation, can effectively improve the simulation efficiency, and thus is a promising approach to large-scale complex manufacturing system simulation (CMSS). Therefore, this paper focuses on large-scale CMSS based on PADS (referred to as simply CMSS hereafter).

A large-scale complex manufacturing system usually contains a large number of entities that need to, on the one hand, carry out complex operations and, on the other hand, interact with the outside frequently. The logistics system of a manufacturing system supply chain typically includes various warehouses, and each warehouse needs to receive/send

✉ Lining Xing
xinglining@gmail.com

1 College of Systems Engineering, National University of Defense Technology, Changsha, China

2 College of Engineering, Shanghai Polytechnic University, Shanghai, China

3 College of Electronic Warfare, National University of Defense Technology, Hefei, China

goods from/to other warehouses. Once a batch of goods arrives, the warehouse needs to perform a series of complex goods-management operations, such as unloading, classification, calculation of the optimal scheduling scheme, and dispatch scheduling. In the simulation of such a complex manufacturing system, each simulation entity[1] needs to perform complex computations and communicate with the outside, placing a high demand on computing resources. Meanwhile, frequent time synchronizations among entities are performed to ensure the causal correctness of simulation results obtained by parallel execution. The efficient execution of such simulation requires sufficient computing resources. A cloud is usually scalable and able to provide unlimited computing resources, and thus deploying CMSS on a cloud is an attractive option. In a cloud environment, computing resources are usually provided for customers in the form of service instances (i.e., virtual machines or nodes in this paper), and multiple service instances are often combined to offer sufficient computing resources. However, the frequent synchronizations among entities in CMSS mean that both the computing power of service instances and the network latency among them highly influence execution efficiency. As a consequence, different service instance combinations (a service instance combination refers to a group of service instances used in one single execution) can result in different running times, and inappropriate computing resource allocation can even lead to low execution efficiency. Hence, it is necessary to allocate proper resources for CMSS for the sake of efficiency. Further, owing to frequent synchronizations, the time consumed by one entity in a synchronization cycle is affected by those of the others, increasing the difficulty of estimating the influence of service instance's computing power and network latency on running time. Consequently, it is challenging to allocate appropriate service instances (i.e., allocating coordinated computing resources) for CMSS to ensure execution efficiency. Hence, it is necessary to investigate service instance allocation for CMSS. Existing methods do not take the frequent synchronizations among entities in CMSS into consideration when allocating service instances, and thus are not suitable approaches. To fill the gap left by current approaches, this paper proposes an intelligent scheduling algorithm for allocating appropriate service instances to reduce CMSS application running time (the time consumed by the execution of the application) in a cloud environment. The main contributions are as follows:

1. A performance estimation model is developed. It can be used to estimate the running time of a CMSS application on a service instance combination.
2. The mutation phase of a genetic algorithm (GA) is adapted for this model. For the selected gene, the adaptation aims to obtain an optimal service instance combination that minimizes the running time.
3. An intelligent scheduling algorithm called the performance estimation model (PEM)-based optimization algorithm (PEMOA) is proposed. It is used to search for the optimal service instance allocation scheme that will minimize the running time of a CMSS application in a cloud environment.
4. A series of experiments were performed that demonstrates the advantage of PEMOA with respect to running time.

This paper is organized as follows. Section 2 reviews the related work. Section 3 first describes the performance estimation model for different service instance combinations and then presents the resource allocation algorithm. Section 4 presents the results of experiments to demonstrate the advantage of the proposed algorithm. Finally, this paper is concluded and future tasks are discussed in Sect. 5.

## 2 Related works

Existing resource allocation schemes consist of dispatching tasks to resources or allocating resources for tasks. The scheduling of tasks usually covers one or multiple objectives. For example, Han et al. [6] studied a scheduling strategy to improve the robustness and stability. Chen et al. [7] proposed an uncertainty-aware scheduling approach to balance cost, deviation, resource efficiency, and fairness. Gong et al. [8] developed an artificial bee colony algorithm with two conflicting objects: minimum of the makespan and maximum of the workload. Further, the allocation of resources for tasks can be divided into the allocation of service instances for tasks and the allocation of hosts for service instances. This paper focuses on allocating service instances for tasks, which has also been used to improve execution efficiency in a cloud environment in some studies. For instance, Mezmaz et al. [9] proposed an algorithm to obtain an resource allocation scheme that minimizes consumed time and energy. Jena et al. [10] proposed a GA-based resource allocation and task scheduling scheme to minimize the execution time and maximize satisfaction. Dam et al. [11] presented an algorithm to minimize execution time while simultaneously minimizing the number of service instances. However, all these methods consider the case in which one task occupies only one service instance. The allocation of multiple service instances for a task has also been studied by many researchers. Wei et al. [12] discussed a scheduling scheme for collaborative tasks in a grid environment and Chen et al. [13] described a computing technique for the collaboration of multiple ser-

[1] Each entity in real system is modeled as a simulation entity. For simplicity, when referring to simulation, a simulation entity is referred to as an entity.

vice instances. Beaumont et al. [14] developed an algorithm to schedule a bag of tasks in a cloud. Nevertheless, these methods are suitable for a task composed of multiple highly independent subtasks among which there are no frequent interactions. During deployment, each subtask is dispatched to a service instance. Essentially, it is similar to allocating a service instance for a task. When allocating service instances for tasks, these methods are more concerned with network bandwidth than network latency. In contrast, in CMSS, entities are synchronized frequently and there are many short messages [15]. Hence, network latency can clearly influence execution efficiency and should be carefully considered during resource allocation. For a task consisting of subtasks with complex interactions, such as massage passing interface (MPI) or work-flow tasks, advance-reservation or work-flow scheduling schemes are utilized in resource allocation [16] [17]. These strategies assume that the time consumed by each subtask remains constant in different service instance combinations, and minimize completion time by changing the times at which subtasks arrive. However, in CMSS, the time consumed by each entity is affected by the network latency and other entities, and thus changes with respect to the service instance combination. In other studies [18], authors proposed a scheduling strategy for co-simulation tasks that comprise multiple federates and deployed these federates to service instances. Considering the frequent interactions among federates, the method migrated federates to as few service instances as possible. However, this method does not take the network latency into account when allocating service instances for tasks.

In summary, current resource allocation strategies are still not suitable for allocating service instances for efficient CMSS execution, and a resource allocation algorithm specifically for this goal is necessary.

## 3 Intelligent resource allocation for CMSS

This section first presents the PEM used to estimate the running time of CMSS on different service instance combinations. Then, an intelligent scheduling algorithm is proposed for searching for the optimal service instance allocation. The proposed algorithm combines simulated annealing (SA) and a GA to obtain the optimal resource allocation scheme. In the proposed algorithm, the PEM is introduced as the fitness function.

### 3.1 Performance estimation model of CMSS

This paper focuses on the CMSS based on PADS, and thus analyzing PADS execution in advance is necessary. Next, the influence of the computing power of service instances as well as the network latency among them on the running time is described, i.e., the PEM is developed to estimate the running time on different service instance combinations.

#### 3.1.1 Analysis of PADS

During a PADS execution, entities are frequently synchronized by time synchronization algorithms that mainly include the optimistic time synchronization algorithm and the conservative time synchronization algorithm. Because the optimistic time synchronization algorithm is less efficient than the conservative time synchronization algorithm [15] in a cloud environment, this study considers only the latter one. In the conservative time synchronization algorithm, each entity repeats four operations: i.e., receive events, execute safe events, update global safe time (GST) and release events, until the simulation ends. The conservative synchronization algorithm is as shown in Algorithm 1, where "end" is the end time of the simulation. During execution, GST is used to detect the termination of the program and determine whether an event is safe. A safe events is one that can be executed safely without incurring rollback. That is, if the timestamp of an event is less than the minimum timestamp of future events, the event is called a safe event.

**Algorithm 1:** Conservative Time Synchronization Algorithm

```
1  while GST <end do
2      //step 1: execute all safe events
3      execute safe events;
4      //step 2: synchronize with other entities and update GST
5      update GST;
6      //step 3: release events to outside
7      release events;
8      //step 4: receive events from outside
9      receive events;
10 end
```

In a distributed execution, a server is required to ensure entities are synchronized. If a PADS application is executed on multiple service instances, a service instance is needed perform the role of the server. Synchronization occurs in two cases: one is when all entities are synchronized to ensure that all sent events have been received to avoid straggler events, and the other occurs when the GST is calculated. During the GST update phase, each entity sends a GST update request to the server and waits for the new GST from the server. When the server has received GST update requests from all entities, it calculates a new GST and then broadcasts the new GST to all entities. This process is shown in Fig. 1, and the details are as follows: (1) All entities execute all their safe events independently. (2) Each entity sends a GST update request to the server and then waits in the GST barrier. (3) After the server receives GST update requests from all entities, it

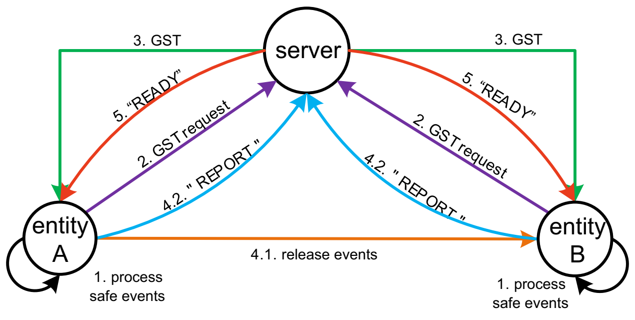

**Fig. 1** Operations in a simulation cycle

calculates a new GST and broadcasts the new GST to each entity. (4) Each entity releases events, and notifies the server of the number of received or released events (this notification is called as a "REPORT" message), and then waits in the event barrier. (5) After all the sent events have been received, the server broadcasts a message to tell the entities to exit the event barrier. This message is called a "READY" message. Then, the next simulation cycle begins. In each simulation cycle, after an entity sends out the GST request or releases all the events targeting outside, it is blocked in the barrier until a new GST or a "READY" message arrives. For convenience, this paper treats the gap between two GST barriers as the time consumed in a simulation cycle, as illustrated in Fig. 2.

As Fig. 2 shows, the time consumed in each simulation cycle is the sum of the time consumed in each phase, as follows:

$$T_i = T_i^g + T_i^s + T_i^r + T_i^e + T_i^u, \tag{1}$$

where, for the $i$th simulation cycle, $T_i$ is the total time elapsed; $T_i^g$ is the duration from the time at which the server sends out the GST to the time at which the entity receives GST; $T_i^s$ is the duration from the time at which the entity sends out events to the time at which the server receives the "REPORT" message; $T_i^r$ is the duration from the time at which the server sends out "READY" messages to the time at which the entities receive them; $T_i^e$ is the time required to execute events; $T_i^u$ is the duration from the time at which the entity sends out a GST update request to the time at which the server receives the GST requests from all the entities in $i$th simulation cycle. As shown in Fig. 2, there are two barriers in each simulation cycle: an event barrier and a GST barrier. Each barrier creates the synchronization of the entities: a fast entity must wait for the slower ones. Therefore, the time consumed in a simulation cycle can be divided into two parts according to the barriers. The first part starts with GST broadcast and ends with the event barrier. This stage mainly consists of communication, and the time it consumes is called the communication cost. The second part lasts from the start of the "READY" broadcast to the end of the GST barrier. This stage mainly consists of event execution, and the corresponding time consumed is called the computation cost. Thus, the costs can be expressed as follows: $T_i^p = T_i^r + T_i^e + T_i^u$ and $T_i^m = T_i^g + T_i^s$, where $T_i^m$ is the communication cost in the $i$th simulation cycle and $T_i^p$ is the computation cost in the $i$th simulation cycle. Then, $T_i = T_i^p + T_i^m$, and the execution time of application $T$ can be expressed as follows, where $C$ is the number of simulation cycles.

$$T = \sum_i^C T_i = \sum_i^C (T_i^p + T_i^m). \tag{2}$$

To estimate $T_i$, the key is to determine the number of simulation cycles and estimate both the computation and communication costs.

### 3.1.2 Running time estimation

According to formula (2), the first step in estimating the running time is to obtain the number of simulation cycles. This section describes how to obtain the GST sequence using collected events. Using the GST sequence, the number of simulation cycles can be obtained because two adjacent simulation cycles are separated by a GTS. Once the GST sequence is known, the processed events in each simulation cycle can be acquired as well. Then, the communication and computation costs for different service instance combinations can be determined, based on which the time consumed for a service instance combination can be estimated.

#### Obtaining the GST sequence based on executed events

Obtaining the GST sequence, in this paper, depends on the collected event information. This information should include the event id, entity id, timestamp, and consumed time. Each event has a unique id, so does the entity. The timestamp represents the simulation time at which the event is executed, and the consumed time is the time consumed by event execution. To obtain the GST sequence according to the executed events, the internal structure of an entity should be studied in advance.

Each entity consists of a local queue, an input queue, and an output queue, as shown in Fig. 3. The input queue is used to store events received from outside, the local queue is used to store events that the entity schedules itself, and the output queue is used to store events scheduled for the outside. During execution, each entity maintains a local safe time (LST), which denotes the minimum timestamp of future events and can be calculated according to the GST. In each entity, events with a timestamp less than its LST are treated as safe events and can be executed. After completing the execution of the safe events, new events are generated and are pushed into the

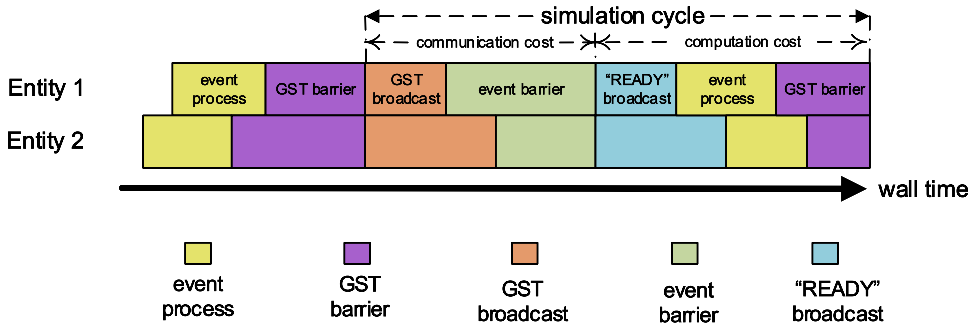

**Fig. 2** Consumed time in a simulation cycle

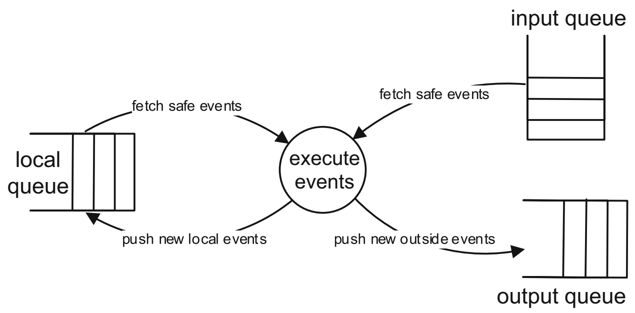

**Fig. 3** Internal structure of a simulation entity

output queue or the local queue. Afterwards, the new GST is calculated as the minimum timestamp of the events existing in the output queue, local queue, and input queue of all entities. Then, the next simulation cycle begins. Obviously, GST calculation needs to be combined with LST calculation of each entity.

The analysis above indicates that newly generated events by the execution of an event must be obtained. Thus, it is necessary to use the event information to establish a scheduling relationship among the events. The procedure for calculating the GST is shown in Algorithm 2. First, for each entity, a corresponding class "simEntity" is constructed and stored in list "enlist". Class "simEntity" contains six variables: "id", "lst", "minTime", "inputQueue", "localQueue", and "outputQueue". Variable "id" is the entity id, and "lst" represents LST, and "minTime" is the minimum timestamp of events. The "inputQueue", "localQueue" and "outputQueue" are used to represent the local, input and output queues respectively as shown in Fig. 3. Then, for each entity, its LST is updated according to last LST, as lines 6–9 show. Here, $LA$ is the lookahead used in CMSS. The "lastLSTList" stores the last LST for all entities. Line 8 selects the value of GST or LST, whicherver is larger, so that more events can be processed safely. Then, each entity executes its safe events, and the events newly generated according to the scheduling relationship are put into "localQueue" or "outputQueue". Next, "lst" is saved to update the LST, the "minTime" is calculated, and the GST is updated to the minimum of the "minTime" of all entities after they have completed safe event execution, as lines 11–13 show. Afterwards, each entity releases the events in "outputQueue", and they are added to the "inputQueue" of the destination entities, as lines 16–18 show. Finally, "lastLSTList" is updated and the GST is stored, as lines 19 and 20 show. Then, the next round starts. Through this method, the GST sequence can be obtained according to the collected events, and the gap between two GSTs is a simulation cycle. Moreover, the events processed in each simulation cycle can be obtained, and the time consumed by event execution can be calculated.

**Communication cost estimation** According to the definition of $T_i^m$, the communication cost is the time consumed from the start of GST broadcast to the end of the event barrier. When an entity receives a new GST, it releases events to the outside, transmits a red "REPORT" message to the server, and then is blocked in the event barrier. Thus, the communication cost includes the overhead caused by transmitting a GST from the server to all entities, transmitting the events released from the source entity to the destination entity, and transmitting a "REPORT" message to the server when an entity receives events, i.e., the overheads in steps 4.1, 4.2, and 3 in Fig. 1. While the entities wait for a "READY" message from the server, they are blocked in the event barrier, but the server broadcasts the "READY" message only when it receives a "REPORT" message indicating that all the sent events have been received. Therefore, the communication cost in each simulation cycle is the maximum of the sum of these three overheads. To account for the overhead caused by some other basic operations performed at this stage, such as sending or receiving a message in the server, the method proposed in this paper includes an additional variable that represents their sum of them. Using this method, the time consumed during this stage can be estimated by following formula.

$$T_i^m = \max_{1 \le j,k \le n} (O_k + O_{jk} + O_j) + T_i^a, \quad (3)$$

Here, $O_k$ is the overhead of transmitting a message from the server to entity $k$, $O_{jk}$ is the overhead from entity $j$ to entity

**Algorithm 2:** calculation of GST based on collected events

```
GSTList=0; lastLSTList=0; $GST$=0;
while $GST$ <tend do
    LSTList=0;
    foreach entity en in enList do
        $en.lst = en.minTime = \infty$;
        foreach lastLst in lastLSTList do
            $en.lst$ = min($en.lst, lastLst + LA$);
            $en.lst$ = max($en.lst, GST$);
        end
        $en$.executeSafeEvents();
        LSTList[$en.id$] = $en.lst$;
        $en.minTime$ = minimum timestamp of events;
        $GST$ = min($GST, en.minTime$);
    end
    //release events
    foreach element en in enList do
        $en$.releaseEvents();
    end
    copy LSTLst to lastLSTList;
    GSTList.add(GST);
end
```

$k$, $T_i^a$ is the time consumed by additional operations in the $i$th simulation cycle, and $n$ is the number of entities.

According to the LogP model [19], the overhead of transmitting a message of size $L$ from one terminal to another is $O = O_{null} + k * L$, where $O_{null}$ is the overhead of transmitting a null message and $k$ is a coefficient that denotes the overhead of handling a message of unit size. In PADS, messages are usually a few bytes long [15], and thus this paper assumes $O \approx O_{null}$. Further, the overhead of transmitting a null message mainly includes the sending, propagation, and receiving overheads, of which the sending and receiving overheads are much lower than the propagation overhead. Therefore, the proposed method mainly considers the propagation overhead when estimating $O_{null}$. To include the influence of other auxiliary information transmissions that ensure the correctness of the simulation execution in each simulation cycle, the proposed model integrates a factor in the transmission overhead calculation for a message (which can be treated as a null message because it is usually short) and assumes $O \approx \lambda D_{sd}$, where $D_{sd}$ is the network delay from the source to the destination and $\lambda$ is a coefficient that is determined by the simulation engine and can be obtained through linear fitting.

**Computation cost estimation** According to the definition of $T_i^p$, the computation cost includes the overhead caused by transmitting a “READY” message from the server to the entity, executing the safe events, and transmitting a GST update request to the server. Each entity starts to execute events when it receives a “READY” message from the server. Upon completing the event execution, the entity sends a GST update request to the server, and then is blocked in the GST barrier while waiting for new GST, as shown in Fig. 1. At this stage, the new GST will not be broadcast until the server receives the GST requests from all entities, and all entities will exit GST barrier only when they receive the new GST. Hence, the computation cost in each simulation cycle is the maximum of the sum of these three overheads above, as follows.

$$\begin{aligned} T_i^p &= \max_{1\le j\le n} (O_j + T_{ij}^e + O_j) \\ &= \max_{1\le j\le n} (2 * O_j + T_{ij}^e), \end{aligned} \tag{4}$$

where $O_j$ is the overhead of transmitting a message from the server to entity $i$, $T_{ij}^e$ is the time consumed by executing events of the $j$th entity in the $i$th simulation cycle, and $n$ is the number of entities.

Considering that the service instances in a cloud environment are homogeneous, their different computing powers mainly influence the time consumed by events. If the computing power of a service instance A and the time consumed by an event on this service instance are known, then the time consumed by the event on another service instance B can be estimated if the computing power of service instance B is known. That is, given two service instances $SI_i$ and $SI_j$, and $W_i$, the computing power of $SI_i$, and $W_j$, the computing power of $SI_j$, if the execution time of event $E$ on service instance $SI_i$ is $T_i$, then the execution time $T_j$ of event $E$ on the service instance $SI_j$ is:

$$T_j = (W_i * T_i)/W_j. \tag{5}$$

Formulas (2)–(5) are combined to form the PEM that can be used to estimate the running time of CMSS applications on a cloud.

To verify the effectiveness of the PEM, a group of CMSS applications based on the PHOLD model [20], which is used as benchmark in PADS, were implemented. In each event, $10^5$ addition computations were performed. The applications were executed on three, four, and sixe service instances, respectively. The network latency was set using the tc command. Then, the estimated running time based on PEM was determined and compared with the actual running time. The correlation coefficients between the estimated running time and actual running times are 0.988, 0.989, and 0.989, respectively. The high correlation between these values shows that the estimated running time can be used as a guideline for allocating resources to reduce the actual running time.

### 3.2 Optimization via genetic algorithm

Typically, a CMSS application needs to be executed many times and one single execution requires multiple service instances. Assume that a CMSS application needs to be executed $m$ times simultaneously and that a single execution needs $n$ service instances. In this case, the overall running time $T_D$ of executing a CMSS application on a cloud can be expressed as follows.

$$T_D = \max_{1 \le v \le M} T^v, \tag{6}$$

where $T^v$ is the time consumed by the execution on service instance combination $v$ and there are $m*n$ service instances in total that must be chosen from the provided service instances. Formula (6) infers that an appropriate service instance allocation can reduce the running time of a CMSS application on a cloud. To obtain the optimal scheme, this section presents PEMOA, in which SA is incorporated into the mutation phase of a GA, with PEM introduced as the fitness function for assessing a service instance combination.

Here, assume that the service instances in the cloud environment is sufficient. The service instances provided by the cloud service providers are expressed as $P = (P_1, P_2 \cdots P_Q)$, and network latency among them is expressed as a two-dimensional array $D = [delay_{ij}]_{(Q*Q)}$, where $1 \le i, j \le Q$, $Q$ is the number of service instances, and $delay_{ij}$ is the network latency between service instance $i$ and $j$. The service instances required to execute a CMSS application are expressed as $R = (R_1, R_2 \cdots R_{m*n})$, where $m*n$ denotes the number of service instances required by executing a CMSS application in a cloud environment. Usually, a service instance is represented by a triplet $< cp, disk, bw >$, where $cp$, $disk$, and $bw$ represent the computing power, disk capacity, and bandwidth of the service instance, respectively. Therefore, the allocation of service instances can be expressed as follows.

$$Minimize \quad T_D \tag{7}$$

subject to

$$R_i^d \le P_i^d \quad (1 \le i \le m*n), \tag{8}$$
$$R_i^b \le P_i^b \quad (1 \le i \le m*n), \tag{9}$$

where $R_i^d$ and $R_i^b$ denote the disk and bandwidth of the $i$th required service instance. Similarly, $P_i^d$ and $P_i^b$ represent the disk and bandwidth of the $i$th provided service instance. The selection of service instances to minimize the time consumed by a CMSS application is an NP-hard problem. SA and GA are two common algorithms used for searching for an optimal solution to such problems. However, GA is strong at global searches but weak at local searches, and SA is effective for local searches [21]. The proposed method hence combines SA and GA to utilize their respective advantages. The main steps include coding, mutation, and crossover. Except for the mutation phase, in which SA is introduced, the phases are similar to those of GA. The details are as follows.

- Coding: Because a CMSS needs to be executed $m$ times simultaneously and a single execution needs $n$ service instances, a chromosome is expressed as a list of qualified service instances with a length of $m*n$. Each gene of a chromosome is a service instance (SI). A service instance is qualified if it satisfies the requirement of (8) and (9). Then, each chromosome can be expressed as $[SI_{11}, SI_{12} \cdots SI_{1n} \cdots SI_{ij} \cdots SI_{m*n}]$, as shown in Fig. 4. For simplicity, a chromosome is expressed as $[SI_{ij}]_{(m*n)}$, where $1 \le i \le m, 1 \le j \le n, i$ is the service instance combination index, and $j$ is the service instance index in the service instance combination (SIC). Each gene in one chromosome is from the provided service instances (the provided service instances are contained in a set of SIs ) and a SI can be “used” in one chromosome only once. Thus, when selecting a SI for a gene in one chromosome, the “unused” SIs are first determined, and then, a qualified SI is chosen from them. If a SI is selected to constitute a chromosome, it is marked as “used” in this chromosome. In this way, one “unused” SI can be found for each gene.
- Mutation: As shown in Fig. 4, the first step of the mutation phase is to randomly choose a chromosome and clone it. The clone is denoted as $chromo$. Then, a gene $SI_{ij}$ in $chromo$ is randomly selected and the service instance combination that contains $SI_{ij}$ is determined i.e., the $i$th service instance combination $SIC_i$ of $chromo$. Next, according to $SIC_i$, the “unused” SIs, i.e., the SIs that are not contained by $SI_j$ $(1 \le j < i$, or $i < j \le m)$,

are obtained. The third step is to utilize SA to obtain $n$ service instances from the "unused" SIs to replace $SIC_i$ and hence minimize the running time on the $i$th service instance combination of $chromo$. After mutation, the clone is added into the chromosome set, and the chromosome with the least fitness is removed from the chromosome set.

– Crossover: Two chromosomes are chosen and cloned. The clones are called $chromo_a$ and $chromo_b$. Then, the positions where the crossover starts and ends, expressed as $st$ and $end$, are determined. Next, each pair $< chromo_a[i], chromo_b[i] >$, where $st \leq i \leq end$, are checked to determine if they are interchangeable. Here, $chromo_a[i]$ and $chromo_b[i]$ represent the $i$th gene of $chromo_a$ and $chromo_b$, respectively. If they are interchangeable, the $i$th gene of $chromo_a$ is replaced with $chromo_b[i]$ and the $i$th gene of $chromo_b$ is replaced with $chromo_a[i]$, as shown in Fig. 4. If they are not interchangeable, this pair is skipped. A pair $< chromo_a[i], chromo_b[i] >$ are interchangeable if $chromo_b[i]$ is not contained in $chromo_a$ and $chromo_a[i]$ is not contained in $chromo_b$. After crossover, $chromo_a$ and $chromo_b$ are added to the chromosome set, and the two chromosomes with least fitness are removed from the chromosome set.

The operations above, which form the intelligent resource allocation algorithm for CMSS, are listed in Algorithm 3.

In Algorithm 3, the function calcAverageFitness determines the average fitness of all the chromosomes and the calcMaxFitness obtains the maximum fitness of all the chromosomes. The value of $iteration$ is set by the user to denote the number of iterations. When the loop has been executed a specified number of times, the algorithm is terminated and the best chromosome is the final result. To avoid premature convergence, a self-adaptive strategy is adopted [22]. In this algorithm, the strategy adjusts the mutation probability and crossover probabilities adaptively, as shown in lines 15 and 21. When all the chromosomes converge to a local optimum, i.e., when the difference between the maximum and average fitnesses is similar, both the mutation and crossover probabilities increase. In other words, both mutation and crossover probability vary inversely according to the difference between the maximum and average fitness. Moreover, this algorithm is designed to protect the "good" chromosomes. The closer the fitness of selected chromosomes are to the maximum fitness, the smaller both the mutation and crossover probabilities should be. Therefore, the values of the mutation probability and crossover probability vary in direct proportion to the difference between the fitness of the selected chromosomes and the maximum fitness. Mutation and crossover are performed according to the mutation and crossover probabilities, respectively. In addition, in the mutation and crossover phases, all the operations are performed on the clones so that the good chromosomes are preserved. Using this method, global convergence can be achieved [23]. During the calculation of the fitness of one chromosome, the estimated running time (i.e., the maximum of the result of PEM on the $m$ service instance combinations) is utilized. The fitness is the inverse of the running time for this chromosome. Finally, based on the output of Algorithm 3, the best chromosome is obtained and its contained $m * n$ service instances are used as the allocation scheme of PEMOA.

**Algorithm 3:** Intelligent resource allocation algorithm

```
   Output: bestChromo
1  Init_pop(); //initialize the population
2  preAverageFitness = calcAverageFitness();
3  //iteration is set by user
4  while iteration >0 do
5      curAverageFitness = calcAverageFitness();
6      //function abs is to obtain absolute value
7      preAverageFitness = curAverageFitness;
8      maxFitness = calcMaxFitness();
9      switch PHASE do
10         case MUTATION
11             //fitness is fitness value of selected solution.
12             fitness = getFitness(selectOne());
13             pro_mutation =
               $k_1 * (maxFitness - fitness)/(maxFitness - curAverageFitness)$;
14             mutate(pro_mutation);//perform muatation
               operation
15         endsw
16         case CROSSOVER
17             //fitness is the larger fitness value of the two
               parents to be crossed.
18             fitness = getLargerFitness(selectTwo());
19             pro_crossover =
               $k_2 * (maxFitness - fitness)/(maxFitness - curAverageFitness)$;
20             crossover(pro_crossover);//perform crossover
               operation
21         endsw
22     endsw
23     iteration = iteration-1;
24 end
25 bestChromo = getBestChromosome();
```

## 4 Experiments and analysis

### 4.1 Experiment design

Currently, gang scheduling is often used to schedule parallel tasks. A gang represents the resources required by an application, i.e., a service instance combination for a CMSS application. Because CMSS often needs to be executed multiple times simultaneously, multiple gangs are required. For a bag of gangs, an adapted first come first served (AFCFS)

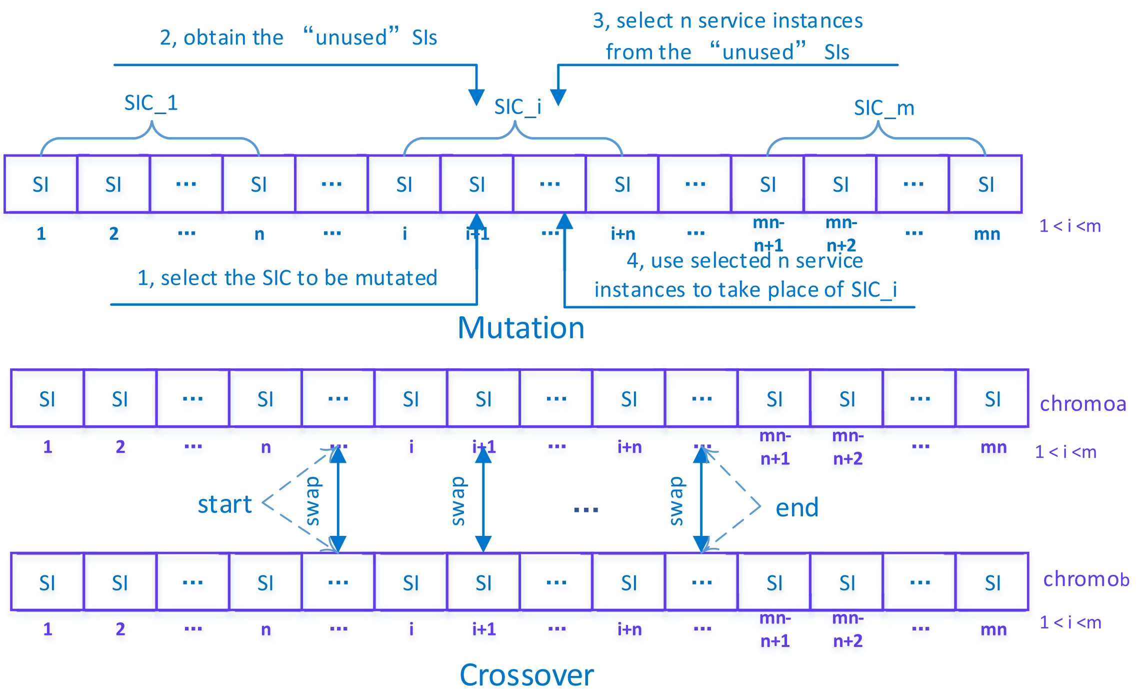
2, obtain the "unused" SIs
3, select n service instances from the "unused" SIs
SIC_1
SIC_i
SIC_m
SI
1
2
n
i
i+1
i+n
mn-n+1
mn-n+2
mn
1 < i <m
1, select the SIC to be mutated
4, use selected n service instances to take place of SIC_i
Mutation
chromoa
start
swap
end
chromob
Crossover

**Fig. 4** Adaptation of genetic algorithm

scheme is often adopted [24]. In addition, a max-min-based algorithm (MMBA) [25] is often used to allocate service instances in a cloud environment to minimize the running time. This paper compares PEMOA with AFCFS and MMBA.

The PHOLD model [20] is usually used as a benchmark in PADS to model the computation within and communication among entities, and is suitable for studying manufacturing systems, which have entities that need to perform complex computations and communicate with the outside frequently. To verify the performance of PEMOA, a PHOLD model and a logistics system model (LSM) for the supply chain were adopted and a series of experiments with different scenarios were performed. In the experiments, the entity groups do not vary, i.e., the entities contained in each group do not vary. Each entity group, as a whole, is dispatched to a service instance before the simulation execution. In the PHOLD model, there are multiple objects. Initially, each object schedules a certain number of events. In each event, a specified number of addition operations are handled and a few events are scheduled. As with an LSM, it is utilized to study the transportation strategy in a supply chain. In an LSM, each warehouse is modeled as an entity, and the transportation of goods from one warehouse to another is modeled as communication among the entities. Each time goods arrive at a warehouse, some computations such as updating the state of the goods and recalculation of the next station toward their destination are performed. Then, goods with the same next station are packed into a batch and sent out as a whole.

Initially, the goods are generated according to a specified number and randomly assigned to different starting warehouses and destinations. During the execution, the goods are distributed to the entity according to their starting warehouses and each batch of goods is transported only to the adjacent warehouse, as shown in Fig. 5. In the experiments, the simulation engine was YH-SUPE [26] and the corresponding value of $\lambda$ was about 6 (measured through linear fitting). The simulation applications were executed on a cluster in which each node was used as a service instance. The operating system of the service instances was Linux Server release 5.5, and each service instance had 4G RAM, which was more than required by the applications above. All the service instances were homogeneous and only the CPU frequency was adjusted. Hence, the computation power of a service instance can be represented by its CPU frequency. The distance between service instances was generated randomly within the range 0 to 10ms. The CPU frequency and network latency of service instances could be adjusted using the cpufrequtils package [27] and tc command. Here, both $k_1$ and $k_2$ in Algorithm 3 were set to 0.5 and 1.0, respectively, according to practical considerations [28]. The variable $iteration$ was set to 1,000. The initial population size in the GA was set to 40. For the SA, the initial temperature was 100 and initial size was 50. The rate of temperature decline is 0.999 and the termination temperature was 1. In the experiments, each application for a resource allocation scheme was executed ten times and the average running time is reported.

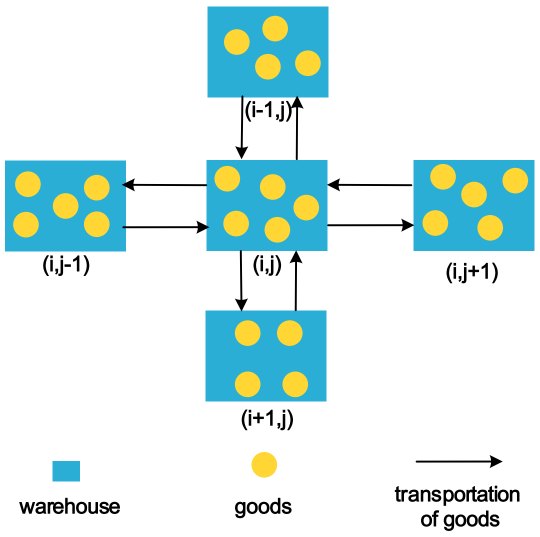

**Fig. 5** Simplified model of the transportation in supply chain

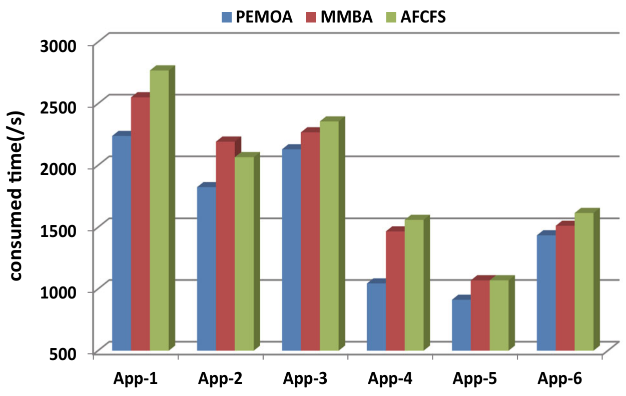

**Fig. 6** Comparison of running time on allocated service instances by three strategies

## 4.2 Performance analysis

Using the above two prototypes, a series of applications were designed and executed on different numbers of service instances (i.e., different sizes of service instance combinations), as shown in Table 1. For each application, 40 copies were simultaneously run on different service instance combinations. An explanation of each variable is given in Table 2. "AppName" is the name of the designed CMSS application in subsequent experiments. For each designed application, the three strategies mentioned above (MMBA, AFCFS, and PEMOA) were used to search for the optimal service instance allocation scheme. The average running times of ten executions on the allocated service instances obtained by the three strategies are shown in Fig. 6. Compared with MMBA and AFCFS, PEMOA can reduce the running time by more than 7% (or even by 35% for some applications). Tt is not certain whether AFCFS or MMBA performs better than the other, but, in general, MMBA outperforms AFCFS. This is because they both select service instances randomly, but AFCFS selects the qualified ones among all the service instances whereas MMBA selects service instances among those with greatest computing power. The computing power is an important factor that influences the running time. Hence, MMBA is better overall, but this cannot be guaranteed in all cases because both of methods ignore the distance among service instances, which is also important. Nevertheless, in PEMOA, both the computing power and distances between service instances are considered, and thus PEMOA can find a better service instance allocation scheme than both AFCFS and MMBA.

**Table 1** Configuration of experiments based on PHOLD and LSM

| Application prototype | PHOLD | | | Application prototype | LSM | | |
|---|---|---|---|---|---|---|---|
| AppName | App-1 | App-2 | App-3 | AppName | App-4 | App-5 | App-6 |
| NO | 6 | 8 | 10 | NG | 50 | 200 | 200 |
| NE | 8 | 10 | 10 | NW | 25 | 50 | 100 |
| NAO | 1.00E4 | 1.00E6 | 1.00E7 | NSI | 2 | 5 | 6 |
| NSI | 3 | 4 | 6 | SET | 10000 | 5000 | 5000 |
| SET | 10000 | 10000 | 10000 | LA | 3 | 2 | 1 |
| LA | 1 | 2 | 3 | | | | |

**Table 2** Explanation of variables

| | | | |
|---|---|---|---|
| NO | Number of objects | NG | Number of goods |
| NE | Number of events | NW | Number of warehouses |
| NAO | Number of addition operations | SET | Simulation end time |
| NSI | Number of service instances | LA | Lookahead |

**Table 3** Configuration of experiments for analyzing influence of computation load

| Application prototype | PHOLD | | | | | | |
|---|---|---|---|---|---|---|---|
| NO | 8 | | | | | | |
| NE | 10 | | | | | | |
| NSI | 4 | | | | | | |
| SET | 5.00E3 | | | | | | |
| LA | 1 | | | | | | |
| AppName | NA-5E4 | NA-1E5 | NA-5E5 | NA-1E6 | NA-5E6 | NA-1E7 | NA-5E7 |
| NAO | 5.00E4 | 1.00E5 | 5.00E5 | 1.00E6 | 5.00E6 | 1.00E7 | 5.00E7 |

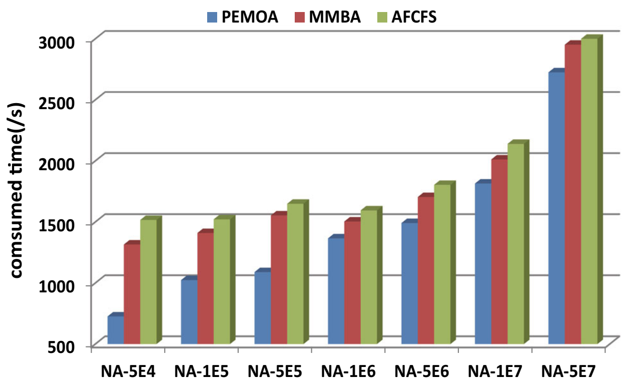

**Fig. 7** Comparison of three strategies for analyzing influence of computation load

Because CMSS applications may be computation intensive or communication intensive, different CMSS applications may place different requirements on the service instance combination. To evaluate the adaptability of PEMOA for different CMSS applications, the impact of the computing load, event density, and number of service instances required on PEMOA performance are considered, and a series of applications with different configurations were designed according to PHOLD. The computation load is the number of operations performed in an event, and the event density is the number of events processed in a simulation cycle. For convenience, the running time on the service instances allocated according to one strategy is called the running time for this strategy.

### 4.2.1 Computation load

This section reports the result of a group of experiments designed to investigate the influence of computation load on the performance of PEMOA, as shown in Table 3. Different values of NAO indicate different computation loads in each event. Figure 7 compares the running time for the service instances allocated by the three strategies. PEMOA allocates service instances that reduce the running time of CMSS applications better than the service instances allocated by MMBA and AFCFS. In addition, as Fig. 7 shows, MMBA is better than AFCFS in most case for various kinds of computation load, which is similar to the conclusion in Sect. 4.2. Moreover, Fig. 7 indicates that, overall, the performance increase obtained by PEMOA compared with those of both MMBA and AFCFS declines as NAO increases. This is because, when NAO increases, each event consumes more time and accounts for a larger proportion of the running time, increasing the impact of computing power on service instance selection in PEMOA (which considers both computing power and network latency). Nevertheless, in MMBA and AFCFS, computing power is the only factor considered in service instance selection. Hence, the impact of computing power of the three strategies becomes closer as NAO increases. As a result, the performance gap between PEMOA and the other two strategies narrows, and the advantage of PEMOA is weakened. In other words, PEMOA performs better for communication-intensive CMSS applications.

**Table 4** Configuration of experiments for analyzing influence of event density

| Application prototype | PHOLD | | | | | | |
|---|---|---|---|---|---|---|---|
| NO | 8 | | | | | | |
| NAO | 5.00E5 | | | | | | |
| NSI | 4 | | | | | | |
| SET | 1.00E4 | | | | | | |
| LA | 3 | | | | | | |
| AppName | NE-5 | NE-10 | NE-15 | NE-20 | NE-25 | NE-30 | NE-35 |
| NE | 5 | 10 | 15 | 20 | 25 | 30 | 35 |

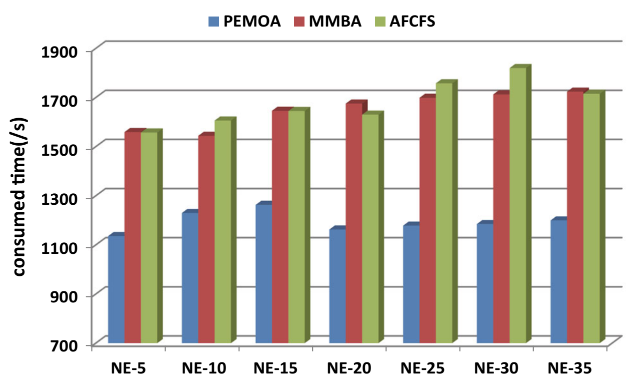

**Fig. 8** Comparison of three strategies for analyzing influence of event density

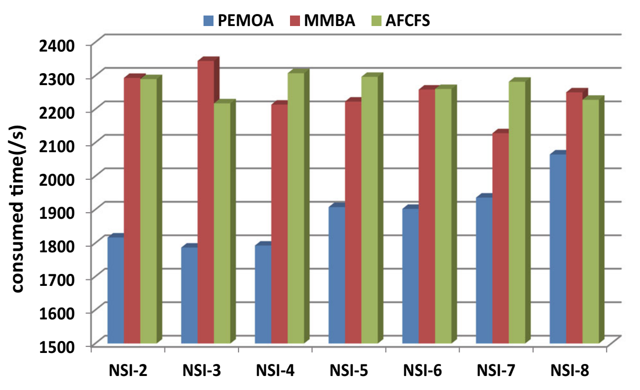

**Fig. 9** Comparison of three strategies for analyzing influence of number of service instances

### 4.2.2 Event density

This section reports the result of a group of experiments designed to investigate the influence of event density (number of events, NE) on the performance of PEMOA, as shown in Table 4. The running times on the service instances allocated by the three strategies are shown in Fig. 8. Compared with MMBA and AFCFS, PEMOA can better reduce the running time of CMSS applications. Moreover, MMBA is better than AFCFS in most cases, which is similar to the results above. In addition, Fig. 8 shows that the number of events has little influence on the performance of PEMOA. This is because, even though the event density increases, the time consumed for event execution does not increase much due the strong computing power of the CPUs. Consequently, the ratio of communication overhead to the time consumed for event execution does not vary much, either. Therefore, and the gap between the performance of PEMOA and those of the other two strategies does not change much when NE increases.

### 4.2.3 Number of service instances

This section reports the result of a group of experiments designed to investigate the influence of the number of service instances (NSI) on the performance of PEMOA, as shown in Table 5. The results, shown in Fig. 9, indicate that the running time for the service instances allocated by PEMOA is less than those for the service instances allocated by MMBA and AFCFS. In addition, the relationship between MMBA and AFCFS is similar to that in above experiments. However, as the size of service instance combinations increases, the relative amount of time saved by PEMOA compared with the results of using MMBA and AFCFS declines. This is because for a CMSS application, fewer copies must be executed simultaneously, and increasing the NSI will result in the selection of more service instances. Given a specific number of service instances, selecting more of them will lead to a less efficient service instance combination. As a result, the gap between PEMOA and other two strategies reduces, as can also be inferred from formula (6). Therefore, for an application that requires fewer service instances, the PEMOA yields greater improvements.

## 5 Conclusion and future works

Simulation is an important stage in a manufacturing system. It can contribute to the design and optimization of the manufacturing system and should be executed as quickly as possible to shorten the time to market. For large-scale CMSS operated in a cloud environment, its execution usually

**Table 5** Configuration of experiments for analyzing influence of number of service instances

| Application prototype | PHOLD | | | | | | |
|---|---|---|---|---|---|---|---|
| NO | 16 | | | | | | |
| NE | 10 | | | | | | |
| NAO | 1.00E6 | | | | | | |
| SET | 1.00E4 | | | | | | |
| LA | 2 | | | | | | |
| AppName | NSI-2 | NSI-3 | NSI-4 | NSI-5 | NSI-6 | NSI-7 | NSI-8 |
| NSI | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

requires multiple service instances to obtain sufficient computing resources, but the computing power and the network latency of the service instances greatly affect the running time, making the appropriate allocation of service instances necessary. However, the frequent synchronizations in CMSS lead to entity lifetime interdependencies, which makes the estimation a service instances computing power and network latency on the running time challenging. Hence, it is difficult to appropriately allocate service instances for CMSS. To solve this problem, this study proposed a PEM based on executed events and the synchronization algorithm for a service instance combination. Then, based on PEM, a resource optimization algorithm called PEMOA that incorporates SA into a GA was proposed to allocate appropriate service instances for reducing CMSS running time in a cloud environment. The experimental results show that PEMOA can reduce the running times of the AFCFS scheme and MMBA by more than 7%. In addition, the influence of event density, computation load, and the number of service instances was analyzed. The results indicate that event density does not affect the performance of PEMOA relative to those of MMBA and AFCFS, but the relative performance declines when either the computation load or the size of the service instance combinations increase. Because only the conservative time synchronization algorithm was studied in this paper, the optimistic time synchronization algorithm will be considered in future work. Moreover, in addition to the resource allocation strategy, fault tolerance and task migration are two factors that should also be considered to ensure efficient CMSS execution. Therefore, they are also directions of our future work.

**Acknowledgements** This work was financially supported by the National Natural Science Foundation of China (61702527, 61802422, 61773120).

**Author contributions** FY and TL wrote the paper; YY, LX, ZL and HC revised this paper.

## Compliance with ethical standards

**Conflict of interest** The authors declare no conflict of interest.

## References

1. Xu X (2012) From cloud computing to cloud manufacturing. Robot Comput Integr Manuf 28(1):75–86
2. Schaefer D (2014) Cloud-based design and manufacturing. Springer, Berlin
3. Negahban A, Smith S (2014) Simulation for manufacturing system design and operation: literature review and analysis. J Manuf Syst 33(2):241–261
4. Mourtzis D, Doukas M, Bernidaki D (2014) Simulation in manufacturing: review and challenges. In: International conference on digital enterprise technology—Det, Rio Patras, Greece
5. Heilala J, Vatanen S, Tonteri H, Montonen J, Lind S, Johansson B, Stahre J (2008) Simulation based sustainable manufactuing system design. In: Proceedings of the winter simulation conference
6. Han Y, Gong D, Jin Y, Pan Q (2019) Evolutionary multiobjective blocking lot-streaming flow shop scheduling with machine breakdowns. IEEE Trans Cybern 49(1):184–197
7. Chen H, Zhu X, Liu G, Pedrycz W (2018) Uncertainty-aware online scheduling for real-time workflows in cloud service environment. IEEE Trans Serv Comput (to be published)
8. Gong D, Han Y, Sun J (2018) A novel hybrid multi-objective artificial bee colony algorithm for blocking lot-streaming flow shop scheduling problems. Knowl Based Syst 148:115–130
9. Mezmaz M, Melab N, Kessaci Y, Lee YC, Talbi EG, Zomaya AY, Tuyttens D (2011) A parallel bi-objective hybrid metaheuristic for energy-aware scheduling for cloud computing systems. J Parallel Distrib Comput 71(11):1497–1508
10. Jena T, Mohanty JR (2018) Ga-based customer-conscious resource allocation and task scheduling in multi-cloud computing. Arab J Sci Eng 43(8):4115–4130
11. Dam S, Mandal G, Dasgupta K, Dutta P (2016) Genetic algorithm and gravitational emulation based hybrid load balancing strategy in cloud computing. In: Third international conference on computer, communication, control and information technology (C3IT)
12. Wei G, Vasilakos AV, Zheng Y, Xiong N (2010) A game-theoretic method of fair resource allocation forcloud computing services. J Supercomput 54(2):252–269
13. Chen XJ, Jing Z, Jun-Huai LI (2011) Key technology for multi-virtual machine collaborative computing oriented to path search tasks. Comput Integr Manuf Syst 17(10):2298–2308
14. Beaumont O, Carter L, Ferrante J, Legrand A, Marchal L, Robert Y (2008) Centralized versus distributed schedulers for bag-of-tasks applications. IEEE Trans Parallel Distrib Syst 19(5):698–709
15. Fujimoto RM, Malik AW, Park AJ (2010) Parallel and distributed simulation in the cloud. SCS M&S Mag 3:1–10
16. Netto S, Netto S, Buyya R (2009) Adaptive co-allocation of distributed resources for parallel applications. PhD thesis, University of Melbourne, Department of Computer Science and software Engineering
17. Chen H, Zhu X, Qiu D, Liu L, Du Z (2017) Scheduling for workflows with security-sensitive intermediate data by selective tasks duplication in clouds. IEEE Trans Parallel Distrib Syst 28(9):2674–2688
18. Yang C, Chai X, Zhang F (2012) Research on co-simulation task scheduling based on virtualization technology under cloud simulation. Springer, Berlin
19. Culler D, Karp R, Patterson D, Sahay A, Schauser KE, Santos E, Subramonian R, Von Eicken T (1993) Logp: towards a realistic model of parallel computation. ACM Sigplan Not 28(7):1–12
20. Park EJ, Eidenbenz S, Santhi N, Chapuis G, Settlemyer B (2015) Parameterized benchmarking of parallel discrete event simulation systems: communication, computation, and memory. In: Winter simulation conference
21. Li WD, Ong SK, Nee AYC (2002) Hybrid genetic algorithm and simulated annealing approach for the optimization of process plans for prismatic parts. Int J Prod Res 40(8):1899–1922
22. Han Y-Y, Gong D, Sun X (2015) A discrete artificial bee colony algorithm incorporating differential evolution for the flow-shop scheduling problem with blocking. Eng Optim 47(7):927–946
23. Liang Y, Leung K-S (2011) Genetic algorithm with adaptive elitist-population strategies for multimodal function optimization. Appl Soft Comput 11(1):2017–2034
24. Zafeirios P, Helen K (2015) Scheduling bags of tasks and gangs in a distributed system. In: 2015 international conference on computer, information and telecommunication systems (CITS)

25. Santhosh B, Manjaiah DH (2016) A hybrid avgtask-min and max-min algorithm for scheduling tasks in cloud computing. In: International Conference on Control, Instrumentation, Communication and Computational Technologies
26. Buquan LIU, Yiping YAO, Wang H (2012) On the technology of high-performance parallel simulation. Chin J Electron 21(1):1–6
27. ThinkWiki (2018) How to use cpufrequtils. http://www.thinkwiki.ort/wiki/How_to_use_cpufrequtils. Accessed 8 Aug 2018
28. Srinivas M, Patnaik LM (1994) Adaptive probabilities of crossover and mutation in genetic algorithms. IEEE Trans Syst Man Cybern 24(4):656–667