

# A Genetic Algorithm Based Efficient Static Load Distribution Strategy for Handling Large-Scale Workloads on Sustainable Computing Systems

Xiaoli Wang and Bharadwaj Veeravalli

**Abstract** A key challenge faced by large-scale computing platforms to go green is the effective utilization of energy at the various processing nodes. Most existing scheduling models assume that processors are able to stay online forever. In reality, processors, however, may have arbitrary unavailable time periods. Hence, if we inadvertently assign tasks to processors without considering the availability constraints, some processors would not be able to finish their assigned workloads. Thus all the unfinished workloads need to be reassigned to other available processors resulting in an inefficient time and energy schedule. In this chapter, we propose a novel *processor availability-aware divisible-load scheduling* model. Using this model, we design a time-efficient genetic algorithm based global optimization technique to derive an optimal load distribution strategy. Our experimental results show that the proposed algorithm adapts to minimize the processing time, hence the energy consumption too, by over 60% compared to other strategies.

**Keywords** Divisible load · Release time · Off-line time · Load distribution · Processor availability

## 1 Introduction

Modern large-scale computing platforms, such as networked computing systems and cloud computing, have imminent need to go green since they are severely constrained by energy related issues [1]. This is predominantly due to their heavy utilization of power and cooling resources which results in rapid energy consumption which in turn

---

X. Wang

School of Computer Science and Technology, Xidian University, Xi'an 710071, China  
e-mail: wangxiaoli@mail.xidian.edu.cn

B. Veeravalli (✉)

Department of Electrical and Computer Engineering, National University of Singapore, Singapore 117576, Singapore  
e-mail: elebv@nus.edu.sg

© Springer International Publishing AG 2017

A.K. Sangaiah et al. (eds.), *Intelligent Decision Support Systems for Sustainable Computing*, Studies in Computational Intelligence 705, DOI 10.1007/978-3-319-53153-3\_2

imparts large carbon footprints on the environment [2]. Emerging sustainable computing technologies, which primarily aim at reducing massive energy consumption by developing certain ab initio computational and mathematical models, methods, and tools for resource allocation and task scheduling, are therefore gain significant interest to researchers and practitioners.

One of the key techniques to save energy is Dynamic Voltage Scaling (DVS). It exploits the hardware characteristics to save energy by degrading CPU voltage and operating frequency while keeping the processor to operate at a slow speed [3]. During the past decades, substantial energy-efficient scheduling strategies have been proposed for DVS-enabled systems [4–7]. These DVS-based techniques, however, may not applicable to virtualized environments where physical processors are shared by multiple virtual machines (VMs) as lowering the supply voltage will inadvertently affect the performance of VMs belonging to different applications [8]. Another promising approach to conserve energy is turning off idle computing nodes in a data center by packing the running VMs to as few physical servers as possible, often called VM consolidation [9]. However, live VM migration must be guaranteed and resources must be properly allocated in order to avoid severe performance degradation due to resource competition by co-located VMs. There are a large amount of studies on the migration strategies, concerning the issues of where, when, and how a VM should be migrated [10–13]. At the current stage, several management issues about VM consolidation still deserve additional investigations. For example, transferring large-sized data over the shared network link is a huge challenge, especially when several goals in terms of Service-Level Agreement (SLA) violation avoidance, minimum communication delay, high system throughput, and high quality of services have to meet [14].

While significant advancements have been made to minimize the energy consumption for sustainable computing, even stronger effort is needed to promote the effective utilization of energy at the various processing nodes. By “effective utilization” we mean that efforts need to be devoted to making compute platforms not just minimizing the energy consumption but also to make every amount of energy consumed for workload computation worthwhile. This is based on the fact that the actual energy consumed for workload computation might not be equal to the energy that is required for workload computation. For example, most existing scheduling models assume that the compute units, which are processors, are able to stay online and available forever. That is to say, it assumes that all processors remain idle at the beginning of the workload assignment and that they will be kept busy until the assigned workload fractions are completed. In reality, processors, however, may have arbitrary unavailable time periods. They may still be busy computing any previous workload even when a new workload arrives and may even get off-line before finish computing the currently assigned load. The time period between release time and off-line time of a processor is referred to as its available time period. Hence, if we inadvertently assign tasks to processors according to their computational capabilities without taking into account of the availability constraints, some processors would not be able to finish their assigned workloads. Thus all of the unfinished workloads need to be reassigned to other available processors resulting in an inefficient

time and energy schedule. Therefore, designing an efficient load distribution strategy seems appropriate when one considers resource (processor) available times.

It is believed that workloads to be scheduled on Heterogenous Sustainable Computing Systems (HSCS) are quite large in size and possess computationally intensive CPU requirements; otherwise, one or a few processors should be enough for workload computation. Also, workloads should be partitionable so that they can simply be further divided into a number of load fractions and distributed to processors for independently parallel computing. Ideally, if a workload can be divided into an arbitrary number of load fractions such that there are no precedence relationships among these fractions, then we refer to it as a divisible load [15]. Actually, divisible loads exist in widely multiple real-world applications, such as real-time video encoding [16, 17], satellite image classification [18], signature searching in a networked collection of files [19], and so on. It may be noted that divisible load modelling can also be adopted for modern day Big Data processing when the requirements of processing demand homogeneous processing on the data.

There are considerable studies available on finding an optimal load distribution strategy for scheduling large-scale divisible loads on various distributed networks with different topologies, including linear networks [20], bus networks [21], tree networks [22], Gaussian, mesh, torus networks [23], and complete b-Ary tree networks [24]. Generally, a load distribution strategy involves two main issues—one in deriving optimal sizes of the workloads to the processors, referred to as an optimal load partition (OLP), and the other is to determine a viable sequence of distribution that achieves minimum processing time, referred to as an optimal load distribution sequence (OLDS).

As for the first issue, in order to obtain a minimized processing time, it is necessary and sufficient to require that all processors stop computing at the same time instant; otherwise, the processing time of the entire workload could be reduced by transferring some load fractions from busy to idle processors. This widely accepted principle in Divisible-Load Theory (DLT) is referred to as the optimality principle, which provides a key to derive a closed-form solution for OLP [25]. However, as mentioned earlier, processors may have arbitrary unavailable time periods in reality. Hence, we could not inadvertently assign tasks to processors according to the optimality principle as usual; otherwise, workload rescheduling would result in an inefficient time and energy schedule. Therefore, searching for an OLP is necessary for sustainable computing where processor available time periods are involved.

As for the second issue, sufficient evidence has shown that load distribution sequences play a significant role in computational performance. For heterogeneous single-level tree networks, it has been proven that only when the load distribution sequence follows the decreasing order of communication speeds does the processing time reach the minimum [26]. As regard to heterogeneous multi-level tree networks, the OLDS depends only on communication speeds of links but not on computation speeds of processors [27]. Nonetheless, the above studies did not consider start-up overheads for both communication and computation into consideration. For the case of homogenous bus networks with start-up overheads, it was shown that the processing time is minimized when the load distribution sequence follows the order in which

the computation speeds of processors decrease [28]. For a large enough workload on heterogeneous single-level tree networks with arbitrary start-up overheads, the sequence of load distribution should follow the decreasing order of the communication speeds in order to achieve minimum processing time [29], but how large a workload should be to consider it as a large enough workload. Moreover, when we consider processor available time periods, does the above conclusion still hold? If not, what sequence does the load distribution should follow to achieve a minimum processing time?

As regard to processor release times alone, several load distribution strategies were proposed for bus networks [30], linear daisy chain networks [31], and single-level tree networks [32], but they did not take start-up overheads and the influence of load distribution sequence into consideration. In order to obtain an OLP and OLDS simultaneously on single-level tree networks with arbitrary processor release times, a *bi-level genetic algorithm* was proposed in [33]. The proposed algorithm comprises two layers of nested genetic algorithms, with the upper genetic algorithm applied for searching an OLDS and the lower algorithm utilized for finding an OLP. However, as the number of processors increases, the proposed bi-level genetic algorithm gets hard to converge. In order to obtain an accurate OLP, an exhaustive search algorithm was proposed in [34] for release-time aware divisible-load scheduling on bus networks, but it did not consider processor off-line times and the influence of OLDS on processing time.

In this chapter, both processor release times and off-line times are explicitly considered in our model, which brings the work more closer to reality. This scheduling problem at hand is complex owing to an inherent nature of the computing platform which could possibly comprise heterogeneous processors. We propose a novel *Processor Availability-Aware Genetic Algorithm (PAA-GA)* based global optimization strategy to minimize the processing time of the entire workload, thus reducing the total energy consumption too, on HSCS.

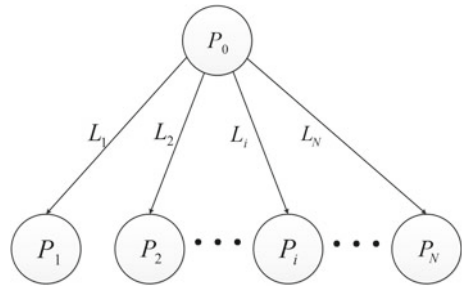
The remaining of this chapter is organized as follows. Section 2 firstly gives a mathematical description of the divisible-load scheduling problem on HSCS with arbitrary start-up overheads and processor available time periods, followed by the proposed availability-aware scheduling model. With this model, we accordingly design algorithm PAA-GA in Sect. 3, which will be evaluated through experiments in Sect. 4. In the last section, conclusions are obtainable.

## 2 Availability-Aware Scheduling Model

### 2.1 Problem Description

An HSCS is considered in this chapter with its topology given in Fig. 1. It comprises  $N + 1$  Heterogeneous processors  $\{P_0, P_1, \dots, P_N\}$  connected through communication links  $\{L_1, L_2, \dots, L_N\}$ , where  $P_0$  signifies the master, while the others

**Fig. 1** An HSCS with  $N + 1$  heterogeneous processors connected in a single-level tree topology



denote worker processors.  $P_0$  does not participate in computation itself but merely takes the responsibility of assigning loads to worker processors.  $P_0$  firstly divides the entire workload  $W_{total}$  into  $N$  fractions  $\vec{A} = (\alpha_1, \alpha_2, \dots, \alpha_N)$  with  $0 \leq \alpha_i \leq W_{total}$  and  $\sum_{i=1}^N \alpha_i = W_{total}$ . Then the load fractions are assigned to worker processors in a certain distribution order  $(P_{\sigma_1}, P_{\sigma_2}, \dots, P_{\sigma_N})$ , where  $\vec{\sigma} = (\sigma_1, \sigma_2, \dots, \sigma_N)$  is processor index which is a permutation of  $(1, 2, \dots, N)$  and  $\alpha_i$  is assigned to processor  $P_{\sigma_i}$  with  $i = 1, 2, \dots, N$ .  $P_0$  sends load fraction to only one processor at a time and each worker starts computing after its entire load fraction has been received completely. Workers cannot communicate and compute simultaneously.

It is necessary to note that, not all worker processors have necessity to participate in workload computation. Suppose that only the first  $n$  processors  $P_{\sigma_1}, P_{\sigma_2}, \dots, P_{\sigma_n}$  in the distribution sequence are needed for workload computation, so they will be assigned with non-zero load fractions, that is,  $\alpha_i > 0$  with  $i = 1, \dots, n$ , while the remaining processors are not assigned with any load fractions, that is, for  $i = n + 1, \dots, N$ ,  $\alpha_i = 0$ .

We consider a heterogeneous system wherein we have, for  $\forall i \neq j$ ,  $w_i \neq w_j$  and  $g_i \neq g_j$ . Also, it is assumed that communication speeds are much faster than computation speeds; otherwise, only one or two processors should be enough to involve in the workload computation [35]. As  $P_0$  assigns  $\alpha_i$  to the  $i$ -th processor  $P_{\sigma_i}$  in the load distribution sequence, the communication and computation components are modelled as affine functions, given by  $e_{\sigma_i} + g_{\sigma_i} \alpha_i$  and  $f_{\sigma_i} + w_{\sigma_i} \alpha_i$ , including communication and computation start-up overheads  $e_{\sigma_i}$  and  $f_{\sigma_i}$  associated with processor  $P_{\sigma_i}$  and link  $L_{\sigma_i}$ , respectively.

Some processors in our system may be engaged in any of the previous workload computation when a new load arrives, say at time  $t = 0$ , so they cannot participate for the newly arrived workload computation until their release times. Meanwhile, they have to finish computing their assigned load fractions before they arrive at their off-line times. It is assumed that processors can estimate their release times by the size of the current workload to process, and the master knows the release and off-line times of all processors. Even though the master does not know the accurate processor off-line times, there exist some prediction techniques to estimate an approximate off-line time for each processor based on a history of processor usage (for more information, please refer to [36–38]). Let  $r_i$  and  $o_i$  be the release time and off-line time of processor  $P_i$  respectively, where  $i = 1, 2, \dots, N$ .

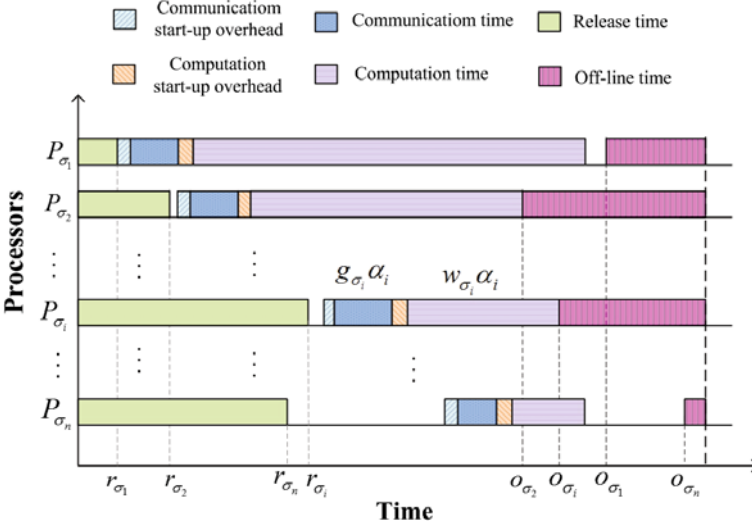


Fig. 2 Gantt chart for load scheduling on HSCS with processor available times

Figure 2 shows a possible Gantt Chart for load scheduling on HSCS with processor release times and off-line times. As illustrated in Fig. 2, the first processor  $P_{\sigma_1}$  starts to receive load fraction  $\alpha_1$  from master  $P_0$  when it gets released at time  $t = r_{\sigma_1}$ . Let  $s_i$  be the start time of processor  $P_{\sigma_i}$ , at which  $P_{\sigma_i}$  starts to receive load fraction  $\alpha_i$  from  $P_0$ . Thus  $s_1 = r_{\sigma_1}$ . Except for  $P_{\sigma_1}$ , the start time  $s_i$  of processor  $P_{\sigma_i}$  depends not only on its release time  $r_{\sigma_i}$ , but also on the start time  $s_{i-1}$  of processor  $P_{\sigma_{i-1}}$  and the communication time ( $e_{\sigma_{i-1}} + g_{\sigma_{i-1}} \alpha_{i-1}$ ) taken by processor  $P_{\sigma_{i-1}}$  to receive its load fraction  $\alpha_{i-1}$  from  $P_0$ . By observing Fig. 2, we obtain that  $s_i = \max \{r_{\sigma_i}, s_{i-1} + e_{\sigma_{i-1}} + g_{\sigma_{i-1}} \alpha_{i-1}\}$ , where  $i = 2, 3, \dots, n$ .

According to the optimality principle of DLT, if processor off-line times are ignored, all processors should finish computing at the same time to obtain a minimized processing time, say at time  $t = T^*$ . Once we consider off-line times, processors whose off-line times are smaller than  $T^*$  will not be able to finish their assigned load fractions. Hence, lest these load fractions be rescheduled and result in a waste of energy consumption, processors should be assigned with appropriate workload sizes according to their available time periods in the first place.

The processing time  $T_i$  of processor  $P_{\sigma_i}$  is given by  $s_i + e_{\sigma_i} + f_{\sigma_i} + (g_{\sigma_i} + w_{\sigma_i}) \alpha_i$ , where  $i = 1, 2, \dots, n$ . It can be observed from the formulation that  $T_i$  depends directly on two parts: the former part  $s_i$  indicating when processor  $P_{\sigma_i}$  starts to receive load from  $P_0$ , and the latter part ( $e_{\sigma_i} + f_{\sigma_i} + (g_{\sigma_i} + w_{\sigma_i}) \alpha_i$ ) representing how long it takes for processor  $P_{\sigma_i}$  to finish computing its assigned load fraction. Both parts are determined directly by load partition  $\vec{A} = \{\alpha_1, \alpha_2, \dots, \alpha_N\}$  and load distribution sequence  $(P_{\sigma_1}, P_{\sigma_2}, \dots, P_{\sigma_N})$ . Hence,  $T_i$  is actually a function of  $\vec{\sigma}$  and  $\vec{A}$ . As the processing time  $T$  of the entire workload lies upon the processor which stops computing the last,

$T$  can be derived as a function of  $\vec{\sigma}$  and  $\vec{A}$  as follows.

$$T(\vec{\sigma}, \vec{A}) = \max_{1 \leq i \leq n} T_i = \max_{1 \leq i \leq n} \{s_i + e_{\sigma_i} + f_{\sigma_i} + (g_{\sigma_i} + w_{\sigma_i}) \alpha_i\}.$$

The objective of divisible-load scheduling on HSCS is to minimize the processing time  $T$  of the entire workload by taking into account processor available time periods, so that every amount of energy is consumed for useful workload computation without wasting, thus reducing the total energy consumption to the utmost. To achieve this goal, one has to determine an optimal load distribution strategy, including an OLDS and OLP. A feasible load distribution strategy should subject to the following four constraints:

- (1) **Workload Constraint:** Each load fraction should be non-negative and not larger than the entire workload, the sum of which is equal to the entire workload. That is to say,  $0 \leq \alpha_i \leq W_{total}$  with  $i = 1, \dots, N$ , and  $\sum_{i=1}^N \alpha_i = W_{total}$ .
- (2) **Processor Constraint:** A load distribution sequence should contain exactly one instance of a processor, without any omission or duplication of a processor or processors. That is,  $\vec{\sigma} = (\sigma_1, \sigma_2, \dots, \sigma_N)$ , where  $\sigma_i \in \{1, 2, \dots, N\}$  with  $i = 1, 2, \dots, N$ ; for  $\forall j, k \in \{1, 2, \dots, N\}$ , if  $j \neq k$ , then  $\sigma_j \neq \sigma_k$ .
- (3) **Participant Constraint:** Not all processors are needed for workload computation. Assuming that only the first  $n$  ( $n \leq N$ ) processors in the distribution sequence are required, we have  $\alpha_i > 0$  with  $i = 1, 2, \dots, n$ , while  $\alpha_i = 0$  when  $i = n + 1, \dots, N$ .
- (4) **Off-Line Time Constraint:** Processors involved in workload computation should stop computing before their off-line time come. That is,  $T_i = s_i + e_{\sigma_i} + (g_{\sigma_i} + w_{\sigma_i}) \alpha_i + f_{\sigma_i} \leq o_{\sigma_i}$  with  $i = 1, \dots, n$ .

## 2.2 A Novel Scheduling Model for Sustainable Computing

Table 1 briefly summarizes related notations and corresponding definitions. In order to solve the scheduling problem mentioned in the previous section, we build a novel processor availability-aware divisible-load scheduling model as follows:

$$\min_{\vec{\sigma}, \vec{A}} T(\vec{\sigma}, \vec{A}) = \min_{\vec{\sigma}, \vec{A}} \left\{ \max_{1 \leq i \leq n} \{s_i + e_{\sigma_i} + f_{\sigma_i} + (g_{\sigma_i} + w_{\sigma_i}) \alpha_i\} \right\}.$$

s.t.

- (1)  $\sum_{i=1}^N \alpha_i = W_{total}$ ,  $0 \leq \alpha_i \leq W_{total}$ ,  $i = 1, \dots, N$ .
- (2)  $\vec{\sigma} = (\sigma_1, \sigma_2, \dots, \sigma_N)$ , where  $\sigma_i \in \{1, 2, \dots, N\}$  and  $i = 1, 2, \dots, N$ .  $\forall j, k \in \{1, 2, \dots, N\}$ , if  $j \neq k$ , then  $\sigma_j \neq \sigma_k$ .
- (3)  $n \leq N$ ;  $\forall i \in \{1, 2, \dots, n\}$ ,  $\alpha_i > 0$ , while  $\forall i \in \{n + 1, \dots, N\}$ ,  $\alpha_i = 0$ .
- (4)  $s_i + e_{\sigma_i} + (g_{\sigma_i} + w_{\sigma_i}) \alpha_i + f_{\sigma_i} \leq o_{\sigma_i}$ ,  $i = 1, \dots, n$ .

**Table 1** Notations and definitions

Notations	Definitions
$W_{total}$	Total size of the entire workload
$N$	Total number of worker processors
$n$	Number of processors required for workload computation
$e_i$	Communication start-up overhead of link $L_i$
$f_i$	Computation start-up overhead of processor $P_i$
$g_i$	Ratio of time taken by link $L_i$ to communicate a given workload to that by a standard link
$w_i$	Ratio of time taken by processor $P_i$ to compute a given workload to that by a standard processor
$\vec{\sigma}$	Processor index used for representing load distribution sequences. $\vec{\sigma} = (\sigma_1, \sigma_2, \dots, \sigma_N)$ is a permutation of $(1, 2, \dots, N)$
$r_i$	Release time of processor $P_i$
$o_i$	Off-line time of processor $P_i$
$s_i$	Start time of the $i$ -th processor $P_{\sigma_i}$ in the distribution sequence
$\vec{A}$	Load partition scheme. $\vec{A} = \{\alpha_1, \alpha_2, \dots, \alpha_N\}$ with each element $\alpha_i$ representing the size of load fraction assigned to the $i$ -th processor $P_{\sigma_i}$ in the distribution sequence
$T$	Processing time of the entire workload
$T_i$	Processing time of the $i$ -th processor $P_{\sigma_i}$ in the distribution sequence

where

(5)  $n = \text{card}(\{\alpha_i \mid \alpha_i \in \vec{A} \text{ and } \alpha_i > 0\})$ , where  $\text{card}(X)$  denotes the number of elements in set  $X$ .

$$(6) \quad s_i = \begin{cases} r_1, & i = 1; \\ \max\{r_{\sigma_i}, s_{i-1} + e_{\sigma_{i-1}} + g_{\sigma_{i-1}} \alpha_{i-1}\}, & i = 2, 3, \dots, n. \end{cases}$$

### 3 Algorithm PAA-GA Based Global Optimization Strategy

In the proposed model, two sets of variables are involved:  $\vec{A} = \{\alpha_1, \alpha_2, \dots, \alpha_N\}$  and  $\vec{\sigma} = (\sigma_1, \sigma_2, \dots, \sigma_N)$ . Therefore, the solution of the proposed model is a mix of real numbers and integer numbers. The problem of deriving an OLDS is similar to Travelling Salesman Problem (TSP) which asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? It is well acknowledged that TSP is an NP-hard problem in combinatorial optimization. Therefore, as an even more complex problem with two sets of variables  $\vec{A}$  and  $\vec{\sigma}$  optimized simultaneously, the problem considered in this chapter is definitely an NP-hard problem. When  $N$  turns out to be large, it is hard to obtain a global optimal solution  $(\vec{A}, \vec{\sigma})$ .



We select Genetic Algorithms (GAs), proposed by Holland [39], to solve our model because GAs have been proven to be a promising technique for task-scheduling problems, especially for complex permutation-based combinatorial optimization problems like TSP [40]. In this chapter, we shall first design an encoding scheme based on the characteristics of the proposed model, on the basis of which genetic operators are introduced, followed by the framework of PAA-GA.

### 3.1 Encoding Scheme

The key point of finding an optimal solution by using GAs is to develop an encoding scheme that can represent the problem to be solved directly and can satisfy the problem constraints easily. In this chapter, a hybrid encoding scheme is adopted. An individual is encoded as  $\vec{I} = (\vec{\sigma}, \vec{A})$ , where  $\vec{\sigma} = (\sigma_1, \sigma_2, \dots, \sigma_N)$  indicates processor index used for representing the load distribution sequence, and  $\vec{A} = (\alpha_1, \alpha_2, \dots, \alpha_N)$  stands for load partition scheme. If  $\alpha_i = 0$ , then it means processor  $P_{\sigma_i}$  does not participate in workload computation.

As a simple example, assume there are six worker processors and that the size of the entire workload is 1000 units. A possible encoding scheme is given as follows:

$$I = \begin{pmatrix} \vec{\sigma} \\ \vec{A} \end{pmatrix} = \begin{pmatrix} \sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6 \\ \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6 \end{pmatrix} = \begin{pmatrix} 2, & 1, & 4, & 6, & 3, & 5 \\ 350, & 200, & 108, & 150, & 120, & 0 \end{pmatrix}.$$

$\vec{\sigma} = (2, 1, 4, 6, 3, 5)$  indicates that the load distribution sequence follows the order of  $(P_2, P_1, P_4, P_6, P_3, P_5)$  and  $\vec{A} = (350, 200, 180, 150, 120, 0)$  means that the sizes of load fractions assigned to processors  $P_2, P_1, P_4, P_6, P_3, P_5$  are 350, 200, 180, 150, 120, 0, respectively. Note that, only 5 processors take part in workload computation since  $\alpha_6 = 0$ , which means  $P_0$  does not assign any load to the last processor  $P_5$  in the distribution sequence, so  $P_5$  does not participate in workload computation. Therefore, we have  $n = 5$ .

### 3.2 Crossover Operators

It is worth noting that given a deterministic load distribution sequence, there exists an OLP that achieves minimum processing time of the entire workload. Hence, the scheduling problem dealt in this chapter can be summarized as determining an OLDS, on the basis of which deriving an OLP to achieve minimum processing time. Therefore, the scheduling problem has actually two layers of decision levels, with the upper level determining an OLDS and the lower level deriving an OLP. With this consideration in mind, we design two crossover operators in this section. One is

to optimize the upper level variable  $\vec{\sigma}$  as well as the lower level variable  $\vec{A}$  simultaneously, while the other is to optimize  $\vec{A}$  alone based on a fixed  $\vec{\sigma}$ .

Also noteworthy is the fact that both  $\vec{\sigma}$  and  $\vec{A}$  are not suitable for traditional two-point crossover. This is because  $\vec{\sigma}$  must contain exactly one instance of a number in  $\{1, 2, \dots, N\}$  and any omission or duplication of numbers leads to an invalid solution. Meanwhile,  $\vec{A}$  has to satisfy that  $\sum_{i=1}^N \alpha_i = W_{total}$ .

### 3.2.1 First Crossover Operator

The first way of crossover is to evolve  $\vec{\sigma}$  and  $\vec{A}$  at the same time. For two parents  $\vec{I}^1 = (\vec{\sigma}^1, \vec{A}^1)$  and  $\vec{I}^2 = (\vec{\sigma}^2, \vec{A}^2)$ , the steps given in Algorithm 1 are adopted to generate two offsprings  $\vec{I}^3 = (\vec{\sigma}^3, \vec{A}^3)$  and  $\vec{I}^4 = (\vec{\sigma}^4, \vec{A}^4)$ .

---

#### Algorithm 1 First Crossover Operator

---

**Input:** Two parents  $\vec{I}^1 = (\vec{\sigma}^1, \vec{A}^1)$  and  $\vec{I}^2 = (\vec{\sigma}^2, \vec{A}^2)$ .

**Output:** Two offsprings  $\vec{I}^3 = (\vec{\sigma}^3, \vec{A}^3)$  and  $\vec{I}^4 = (\vec{\sigma}^4, \vec{A}^4)$ .

- 1: Generate two random integers  $p$  and  $q$  between 1 and  $N$  as the crossover points that satisfy  $1 \leq p < q \leq N$ .
  - 2: **for**  $i = p, \dots, q$  **do**
  - 3:   exchange genes  $\sigma_i^1$  and  $\sigma_i^2$  to obtain  $\sigma_i^3$  and  $\sigma_i^4$ . Let  $\sigma_i^3 = \sigma_i^2$  and  $\sigma_i^4 = \sigma_i^1$ .
  - 4:   exchange genes  $\alpha_i^1$  and  $\alpha_i^2$  to obtain  $\alpha_i^3$  and  $\alpha_i^4$ . Let  $\alpha_i^3 = \alpha_i^2$  and  $\alpha_i^4 = \alpha_i^1$ .
  - 5: **end for**
  - 6: **for**  $i = 1, \dots, p-1$  and  $i = q+1, \dots, N$  **do**
  - 7:   let  $\sigma_i^3 = \sigma_i^1$ ,  $\sigma_i^4 = \sigma_i^2$ ,  $\alpha_i^3 = \alpha_i^1$ , and  $\alpha_i^4 = \alpha_i^2$ .
  - 8: **end for**
  - 9: Establish mapping relationships for interchangeability based on the genes of  $\vec{\sigma}^1$  and  $\vec{\sigma}^2$  between the two crossover points.
  - 10: Based on the mapping relationships, replace the genes of  $\vec{\sigma}^3$  and  $\vec{\sigma}^4$  outside the two crossover points that have the same value with genes inside.
  - 11: Normalize  $\vec{A}^3$  and  $\vec{A}^4$  to ensure that the total size of all load fractions equals the entire workload  $W_{total}$ .
- 

For instance, we have the following two parents.

$$\vec{I}^1 = \begin{pmatrix} \vec{\sigma}^1 \\ \vec{A}^1 \end{pmatrix} = \begin{pmatrix} 2, & | & 1, & 4, & 6, & | & 3, & 5 \\ 350, & | & 200, & 180, & 150, & | & 120, & 0 \end{pmatrix}.$$

$$\vec{I}^2 = \begin{pmatrix} \vec{\sigma}^2 \\ \vec{A}^2 \end{pmatrix} = \begin{pmatrix} 4, & | & 3, & 6, & 2, & | & 5, & 1 \\ 320, & | & 270, & 160, & 140, & | & 150, & 60 \end{pmatrix}.$$

According to Step 1 of Algorithm 1, we first generate two random integers  $p$  and  $q$ . Suppose that  $p = 2$  and  $q = 4$ . After exchanging of genes based on Steps 2–8, we then have,

$$\vec{I}^3 = \begin{pmatrix} \vec{\sigma}^3 \\ \vec{A}^3 \end{pmatrix} = \begin{pmatrix} 2, & | & 3, & 6, & 2, & | & 3, & 5 \\ 350, & | & 270, & 160, & 140, & | & 120, & 0 \end{pmatrix}.$$

$$\vec{I}^4 = \begin{pmatrix} \vec{\sigma}^4 \\ \vec{A}^4 \end{pmatrix} = \begin{pmatrix} 4, & | & 1, & 4, & 6, & | & 5, & 1 \\ 320, & | & 200, & 180, & 150, & | & 150, & 60 \end{pmatrix}.$$

We observe that both  $\vec{I}^3$  and  $\vec{I}^4$  are invalid solutions because some genes of  $\vec{\sigma}^3$  and  $\vec{\sigma}^4$  that are outside the two crossover points have the same values as the genes between the two points, and also that the total size of load fractions is not equal to the entire workload. Therefore, we need to adjust  $\vec{I}^3$  and  $\vec{I}^4$ .

According to Step 9 of Algorithm 1, we establish the following mapping relationships,

$$1 \Leftrightarrow 3, \quad 4 \Leftrightarrow 6, \quad \text{and} \quad 6 \Leftrightarrow 2 \quad \text{implies} \quad 1 \Leftrightarrow 3 \text{ and } 4 \Leftrightarrow 2.$$

With the above mapping relationships, we can fix  $\vec{\sigma}^3$  and  $\vec{\sigma}^4$  now. As for  $\vec{\sigma}^3$ ,  $\sigma_1^3 = 2$  should be replaced by 4,  $\sigma_5^3 = 3$  replaced by 1, and  $\sigma_6^3 = 5$  remaining unchanged. Similarly, as for  $\vec{\sigma}^4$ ,  $\sigma_1^4 = 4$  should be replaced by 2,  $\sigma_5^4 = 5$  remaining unchanged, and  $\sigma_6^4 = 1$  replaced by 3. Hence, we have,

$$\vec{I}^3 = \begin{pmatrix} \vec{\sigma}^3 \\ \vec{A}^3 \end{pmatrix} = \begin{pmatrix} 4, & | & 3, & 6, & 2, & | & 1, & 5 \\ 350, & | & 270, & 160, & 140, & | & 120, & 0 \end{pmatrix}.$$

$$\vec{I}^4 = \begin{pmatrix} \vec{\sigma}^4 \\ \vec{A}^4 \end{pmatrix} = \begin{pmatrix} 2, & | & 1, & 4, & 6, & | & 5, & 3 \\ 320, & | & 200, & 180, & 150, & | & 150, & 60 \end{pmatrix}.$$

After normalization by the last step of Algorithm 1, we obtain two offsprings as follows.

$$\vec{I}^3 = \begin{pmatrix} \vec{\sigma}^3 \\ \vec{A}^3 \end{pmatrix} = \begin{pmatrix} 4, & | & 3, & 6, & 2, & | & 1, & 5 \\ 317, & | & 267, & 158, & 139, & | & 119, & 0 \end{pmatrix}.$$

$$\vec{I}^4 = \begin{pmatrix} \vec{\sigma}^4 \\ \vec{A}^4 \end{pmatrix} = \begin{pmatrix} 2, & | & 1, & 4, & 6, & | & 5, & 3 \\ 321, & | & 183, & 165, & 138, & | & 138, & 55 \end{pmatrix}.$$

### 3.2.2 Second Crossover Operator

The second way of crossover is to evolve  $\vec{A}$  based on a fixed  $\vec{\sigma}$ . Algorithm 2 shows its main steps. As an example, suppose  $p = 2$  and  $q = 5$ . By Steps 2–5 of Algorithm 2, we have,

$$\vec{I}^1 = \begin{pmatrix} \vec{\sigma}^1 \\ \vec{A}^1 \end{pmatrix} = \begin{pmatrix} 2, & | & 1, & 4, & 6, & 3, & | & 5 \\ 350, & | & 200, & 180, & 150, & 120, & | & 0 \end{pmatrix}.$$

$$\vec{I}^2 = \begin{pmatrix} \vec{\sigma}^2 \\ \vec{A}^2 \end{pmatrix} = \begin{pmatrix} 4, & | & 3, & 6, & 2, & 5, & | & 1 \\ 320, & | & 270, & 160, & 140, & 150, & | & 60 \end{pmatrix}.$$

$$\vec{I}^3 = \begin{pmatrix} \vec{\sigma}^3 \\ \vec{A}^3 \end{pmatrix} = \begin{pmatrix} 2, & | & 1, & 4, & 6, & 3, & | & 5 \\ 350, & | & 270, & 160, & 140, & 150, & | & 0 \end{pmatrix}.$$

$$\vec{I}^4 = \begin{pmatrix} \vec{\sigma}^4 \\ \vec{A}^4 \end{pmatrix} = \begin{pmatrix} 4, & | & 3, & 6, & 2, & 5, & | & 1 \\ 320, & | & 200, & 180, & 150, & 120, & | & 60 \end{pmatrix}.$$

It is worth noting that both  $\vec{I}^3$  and  $\vec{I}^4$  are invalid solutions because  $\sum_{i=1}^N \alpha_i^3 < W_{total}$  and  $\sum_{i=1}^N \alpha_i^4 > W_{total}$ . After normalization by Step 6, we obtain two offsprings as follows.

$$\vec{I}^3 = \begin{pmatrix} \vec{\sigma}^3 \\ \vec{A}^3 \end{pmatrix} = \begin{pmatrix} 2, & | & 1, & 4, & 6, & 3, & | & 5 \\ 327, & | & 252, & 150, & 131, & 140, & | & 0 \end{pmatrix}.$$

$$\vec{I}^4 = \begin{pmatrix} \vec{\sigma}^4 \\ \vec{A}^4 \end{pmatrix} = \begin{pmatrix} 4, & | & 3, & 6, & 2, & 5, & | & 1 \\ 311, & | & 194, & 185, & 146, & 116, & | & 58 \end{pmatrix}.$$

---

**Algorithm 2** Second Crossover Operator

---

**Input:** Two parents  $\vec{I}^1 = (\vec{\sigma}^1, \vec{A}^1)$  and  $\vec{I}^2 = (\vec{\sigma}^2, \vec{A}^2)$ .

**Output:** Two offsprings  $\vec{I}^3 = (\vec{\sigma}^3, \vec{A}^3)$  and  $\vec{I}^4 = (\vec{\sigma}^4, \vec{A}^4)$ .

- 1: Let  $\vec{I}^3 = \vec{I}^1$  and  $\vec{I}^4 = \vec{I}^2$ .
  - 2: Randomly select two crossover points  $p$  and  $q$  that satisfy  $1 \leq p < q \leq N$ .
  - 3: **for**  $i = p, \dots, q$  **do**
  - 4:   exchange genes  $\alpha_i^3$  and  $\alpha_i^4$ . Let  $\alpha_i^3 = \alpha_i^2$  and  $\alpha_i^4 = \alpha_i^1$ .
  - 5: **end for**
  - 6: Normalize  $\vec{A}^3$  and  $\vec{A}^4$  to ensure that the total size of all load fractions is equal to that of the entire workload.
- 

### 3.3 Mutation Operator

The purpose of mutation in GAs focuses on preserving and introducing diversity from one generation of a population to the next. It is analogous to biological muta-

tion. Mutation operator alters one or more gene values in an individual from its initial state, thus avoiding local minima and preventing the population from becoming too similar to each other. Mutation occurs to offsprings generated by crossover according to a user-definable mutation probability. This probability should be set low; otherwise, the search will turn into a primitive random search [41].

We apply two-point mutation on both  $\vec{\sigma}$  and  $\vec{A}$  simultaneously to obtain a new offspring. Randomly generate four integers  $p$ ,  $q$ ,  $l$ , and  $m$  that satisfy  $1 \leq p < q \leq N$  and  $1 \leq l < m \leq N$ . Exchange genes  $\sigma_p$  and  $\sigma_q$ , as well as  $\alpha_l$  and  $\alpha_m$ . For example, suppose  $p = 2$ ,  $q = 4$ ,  $l = 1$ , and  $m = 5$ . We obtain offspring  $\vec{l}'$  mutated from  $\vec{l}$  as follows.

$$\vec{l} = \begin{pmatrix} \vec{\sigma} \\ \vec{A} \end{pmatrix} = \begin{pmatrix} 2, & \mathbf{1}, & 4, & \mathbf{6}, & 3, & 5 \\ \mathbf{350}, & 200, & 180, & 150, & \mathbf{120}, & 0 \end{pmatrix}.$$

$$\vec{l}' = \begin{pmatrix} \vec{\sigma}' \\ \vec{A}' \end{pmatrix} = \begin{pmatrix} 2, & \mathbf{6}, & 4, & \mathbf{1}, & 3, & 5 \\ \mathbf{120}, & 200, & 180, & 150, & \mathbf{350}, & 0 \end{pmatrix}.$$

### 3.4 Repair Operator

It cannot be expected that crossover and mutation operators produce new offsprings that satisfy all of the four constraints in our proposed model by default, especially for the Participant Constraint and Off-Line Time Constraint. Thus a newly generated individual need to be checked whether it violates either of the constraints. If so, we have to repair it to a feasible solution. Algorithm 3 gives the main steps of repair operator with the first two steps responsible for Participant Constraint satisfaction and the remaining for Off-Line Time Constraint satisfaction.

Figure 3 shows a Gantt Chart with processors that violate the Off-Line Time Constraint of the proposed model. It can be observed from Fig. 3 that the processing time  $T_2$  of processor  $P_{\sigma_2}$  exceeds its off-line time  $o_{\sigma_2}$ , which results in a time conflict. Therefore, the individual that corresponds to this Gantt Chart violates the Off-Line Time Constraint and needs to be repaired. According to the repair operator, excessive load assigned to  $P_{\sigma_2}$  that causes the time conflict should be removed to its immediate successor  $P_{\sigma_3}$ . Fig. 4 shows a possible Gantt Chart after the load distribution adjustment. It can be seen that  $T_2 = o_{\sigma_2}$ , so the time conflict for processor  $P_{\sigma_2}$  has been eliminated. However, we also notice that the load adjustment gives rise to a new time conflict for processor  $P_{\sigma_3}$ , whose processing time  $T_3$  exceeds its off-line time  $o_{\sigma_3}$ . Therefore, another round of adjustment is required for processor  $P_{\sigma_3}$ . According to the repair operator, excessive load assigned to  $P_{\sigma_3}$  will be scheduled to its immediate successor  $P_{\sigma_4}$ . This process repeats iteratively until there are no time conflicts for all processors.

One may notice that we keep reallocating excessive load from a processor to its successors in the distribution sequence, instead of its predecessors. This is because

---

**Algorithm 3** Repair Operator
 

---

- 1: For a newly generated individual  $\vec{I} = (\vec{\sigma}, \vec{A})$ , calculate the number of processors participating in workload computation by  $n = \text{card}(\{\alpha_i \mid \alpha_i \in \vec{A} \text{ and } \alpha_i > 0\})$ .  
 For an instance, given that

$$\vec{I} = \begin{pmatrix} \vec{\sigma} \\ \vec{A} \end{pmatrix} = \begin{pmatrix} \sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6 \\ \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6 \end{pmatrix} = \begin{pmatrix} 1, 2, 3, 4, 5, 6 \\ 327, 0, 0, 182, 140, 151 \end{pmatrix}.$$

We have  $n = \text{card}(\{327, 182, 140, 151\}) = 4$ .

- 2: Adjust the order of  $(\sigma_i, \alpha_i)$  pairs so that processors with non-zero load fractions are listed in front of the others.

As for the above example, it can be observed that  $\alpha_2 = \alpha_3 = 0$ , so pairs  $(\sigma_2, \alpha_2)$  and  $(\sigma_3, \alpha_3)$  should be placed at the end of  $\vec{I}$ . After reordering, we obtain,

$$\vec{I} = \begin{pmatrix} \vec{\sigma} \\ \vec{A} \end{pmatrix} = \begin{pmatrix} 1, 4, 5, 6, 2, 3 \\ 327, 182, 140, 151, 0, 0 \end{pmatrix}.$$

- 3: Let  $s_1 = r_{\sigma_1}$ .  
 4: **for**  $i = 2, 3, \dots, n$  **do**  
 5:     calculate the start time of each processor by  $s_i = \max\{r_{\sigma_i}, s_{i-1} + e_{\sigma_{i-1}} + g_{\sigma_{i-1}} \alpha_{i-1}\}$ .  
 6: **end for**  
 7: **for**  $i = 1, 2, \dots, n$  **do**  
 8:     compute the processing time of each processor by  $T_i = s_i + (g_{\sigma_i} + w_{\sigma_i}) \alpha_i + e_{\sigma_i} + f_{\sigma_i}$ .  
 9: **end for**  
 10: **for**  $i = 1, 2, \dots, n$  **do**  
 11:     check whether processing time  $T_i$  of processor  $P_{\sigma_i}$  exceeds its off-line time  $o_{\sigma_i}$ . If  $T_i \leq o_{\sigma_i}$  holds for all processors, then stop; otherwise,  $T_i > o_{\sigma_i}$  means too much load has been assigned to processor  $P_{\sigma_i}$  and that excessive load needs to be rescheduled.  
 12: **end for**  
 13: Compute the size of excess load by  $\Delta = (T_i - o_{\sigma_i}) / (g_{\sigma_i} + w_{\sigma_i})$ .  
 14: Assign this excess load fraction  $\Delta$  to the next processor  $P_{\sigma_{i+1}}$  by  $\alpha_{i+1} = \alpha_{i+1} + \Delta$  and  $\alpha_i = \alpha_i - \Delta$ .  
 Then go back to Step 1.
- 

the start time of each processor is determined by the load fractions assigned to its predecessors according to the equation  $s_i = \max\{r_{\sigma_i}, s_{i-1} + e_{\sigma_{i-1}} + g_{\sigma_{i-1}} \alpha_{i-1}\}$ . Suppose processor  $P_i$  violates the Off-Line Time Constraint. If we reallocate excessive load from  $P_i$  to one of the processors before  $P_i$  in the distribution sequence, say  $P_j$ , then the start times of processors behind  $P_j$  may all get postponed, which has a high chance of causing more time conflicts for processors between  $P_i$  and  $P_j$ , thus taking a much longer time for repair operator to fix all time conflicts.

### 3.5 Local Search

As mentioned earlier, searching for an OLDS itself is already an NP-hard problem, not to mention that we also have to search for an OLP. When an HSCS scales up to a large number of processors, it may hard for GAs to converge. In order to improve con-

vergence speed of the proposed GA, we introduce a local search operator. The main idea is to balance load between processors with the longest processing time  $T_{\max}$  and the shortest processing time  $T_{\min}$ , so that all of the processors with processing times not up to their off-line times will eventually stop computing at the same time. The process of the local search operator is given in Algorithm 4.

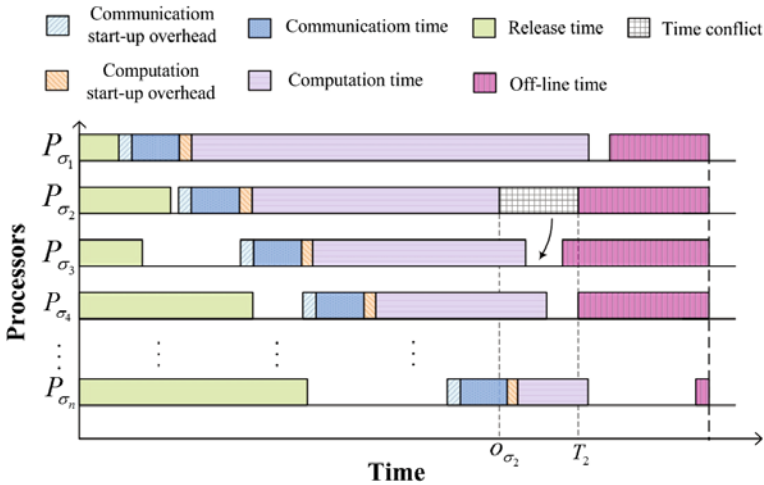
---

**Algorithm 4** Local Search
 

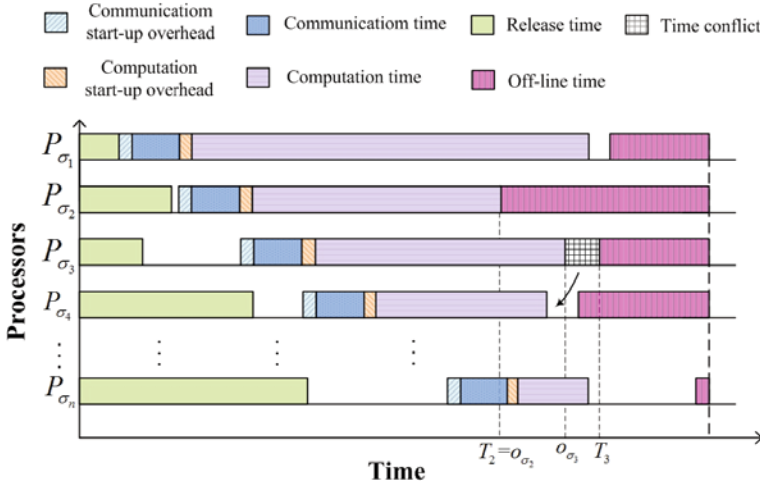
---

- 1: For a given individual  $\vec{I} = (\vec{\sigma}, \vec{A})$ , calculate the number  $n$  of processors participating in the workload computation by  $n = \text{card}(\{\alpha_i \mid \alpha_i \in \vec{A} \text{ and } \alpha_i > 0\})$ .
  - 2: Let  $s_1 = r_{\sigma_1}$ .
  - 3: **for**  $i = 2, 3, \dots, n$  **do**
  - 4:   calculate the start time of each processor by  $s_i = \max\{r_{\sigma_i}, s_{i-1} + e_{\sigma_{i-1}} + g_{\sigma_{i-1}} \alpha_{i-1}\}$ .
  - 5: **end for**
  - 6: **for**  $i = 1, 2, \dots, n$  **do**
  - 7:   compute the processing time of each processor by  $T_i = s_{\sigma_i} + (g_{\sigma_i} + w_{\sigma_i}) \alpha_i + e_{\sigma_i} + f_{\sigma_i}$ .
  - 8: **end for**
  - 9: Among  $P_{\sigma_1}, P_{\sigma_2}, \dots, P_{\sigma_n}$ , find processor  $P_{\sigma_{\max}}$  with the longest processing time  $T_{\max}$  and processor  $P_{\sigma_{\min}}$  with the shortest processing time  $T_{\min}$ . Calculate their time difference by  $\Delta = (T_{\max} - T_{\min})$ .
  - 10: Let  $\beta = (T_{\max} - T_{\min}) / \max\{g_{\sigma_{\max}}, g_{\sigma_{\min}}\}$ . Update individual  $\vec{I} = (\vec{\sigma}, \vec{A})$  by  $\alpha_{\max} = \alpha_{\max} - \beta$  and  $\alpha_{\min} = \alpha_{\min} + \beta$ .
  - 11: Apply repair operator given by Algorithm 3 on the updated individual to ensure that it satisfies all constraints of the proposed model.
- 

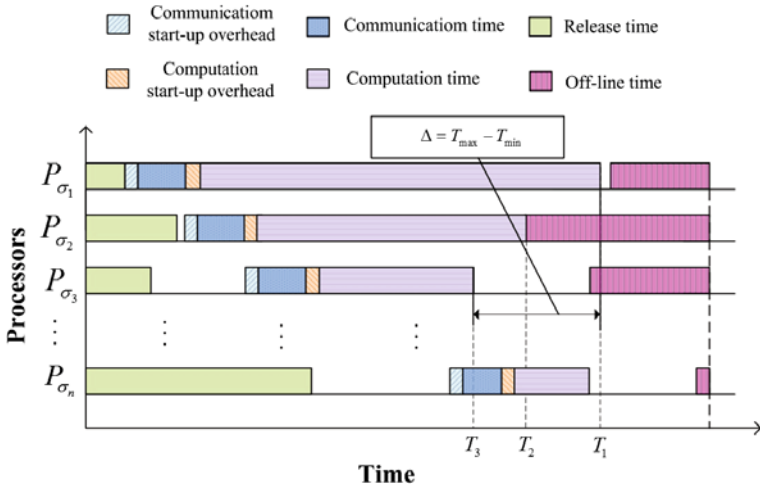
Figure 5 shows a Gantt Chart that corresponds to an individual before applying local search operator. As illustrated in Fig. 5, processor  $P_{\sigma_1}$  has the longest processing



**Fig. 3** Gantt Chart with processors violating the Off-Line Time Constraint of the proposed model



**Fig. 4** Gantt Chart after load distribution adjustment by repair operator



**Fig. 5** Gantt Chart before applying local search

time  $T_1$  and  $P_{\sigma_3}$  has the shortest processing time  $T_3$ . Thus  $P_{\max} = P_{\sigma_1}$  and  $P_{\min} = P_{\sigma_3}$ . After load balancing between  $P_{\sigma_1}$  and  $P_{\sigma_3}$  by local search operator, a possible Gantt Chart is shown in Fig. 6. It can be observed that the time difference between  $T_1$  and  $T_3$  illustrated in Fig. 6 becomes much smaller than that shown in Fig. 5. Hence, the total processing time of the entire workload would be decreased.



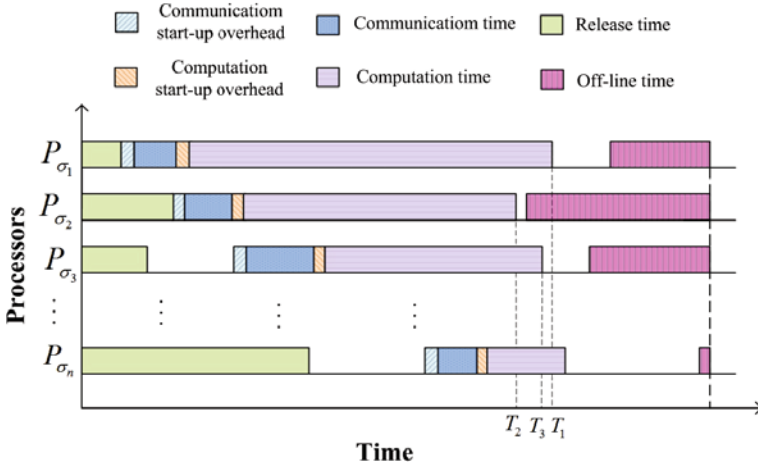


Fig. 6 Gantt Chart after applying local search

### 3.6 Framework of Algorithm PAA-GA

Once encoding scheme is defined, a GA initializes a population of individuals and then improves them through repetitive applications of genetic operators, including crossover, mutation, repair, local search, and selection. The framework and flow chart of algorithm PAA-GA are given in Algorithm 5 and Fig. 7, respectively.

## 4 Experimental Results and Analysis

Several rigours experiments are conducted to study the performance and demonstrate the effectiveness of the proposed algorithm. We employed a compute cluster comprising 15 nodes and the parameters of our HSCS are given in Table 2. In the master node where our proposed scheduling algorithm PAA-GA runs, the following parameters are set: population size  $Popsiz = 100$ , crossover probability  $p_{cros} = 0.6$ , mutation probability  $p_{mut} = 0.02$ , elitist number  $E = 5$ , and stop criterion  $t = 50000$ .

### 4.1 Evaluating the Correctness of PAA-GA

We attempt to make a comparison between an exhaustive algorithm (EA) [34] with our PAA-GA. As expected, although EA will be time-consuming, EA can obtain an absolute minimum processing time. In order to evaluate the correctness of algorithm PAA-GA, we make a comparison between PAA-GA and EA to check whether

---

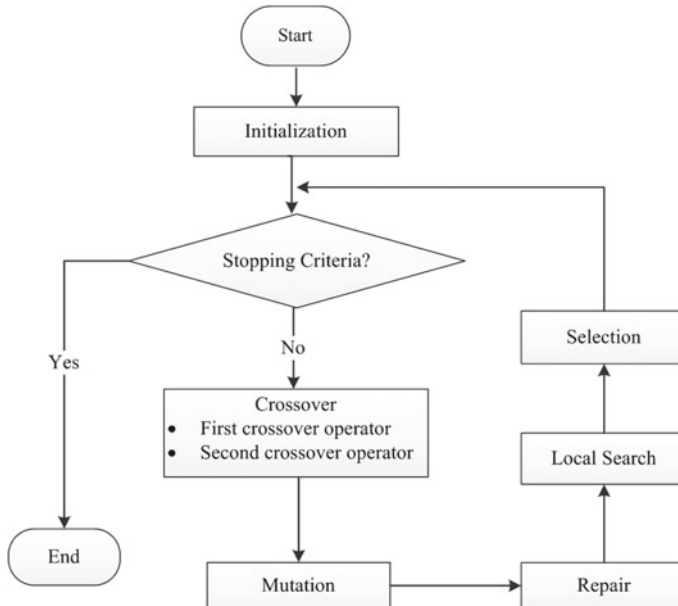
**Algorithm 5** PAA-GA: Processor Availability-Aware Genetic Algorithm
 

---

**Input:** Population size  $Popsiz$ , crossover probability  $p_{cros}$ , mutation probability  $p_{mut}$ , elitist number  $E$  and stop criterion.

**Output:** An OLDS  $\vec{\sigma}$  and OLP  $\vec{A}$ .

- 1: **(Initialization)** Set the population size  $Popsiz$ , crossover probability  $p_{cros}$ , mutation probability  $p_{mut}$ , and elitist number  $E$ . Randomly generate  $Popsiz$  individuals as the initial population  $Pop(0)$  according to the encoding scheme. For each individual  $\vec{I} \in Pop(0)$ , first apply the repair operator given by Algorithm 3 on  $\vec{I}$ , and then compute processing time  $T$  of the entire workload by  $T = \max_{1 \leq i \leq n} \{s_i + e_{\sigma_i} + (g_{\sigma_i} + w_{\sigma_i}) \alpha_i + f_{\sigma_i}\}$ , taking  $1/T$  as the fitness value of  $\vec{I}$ . Let the generation number  $t = 0$ .
  - 2: **(Crossover)** Select  $Popsiz$  individuals into the crossover pool from  $Pop(t)$  by roulette wheel selection. Apply the two crossover operators given by Algorithms 1 and 2 one-by-one on each pair of parents selected from the crossover pool according to crossover probability  $p_{cros}$ . All newly generated offsprings constitute a set denoted by  $O_1(t)$ .
  - 3: **(Mutation)** Apply mutation operator on each of the selected individuals from  $O_1(t)$  according to mutation probability  $p_{mut}$ . All newly generated offsprings constitute a set denoted by  $O_2(t)$ .
  - 4: **(Repair)** Apply repair operator given by Algorithm 3 on each individual in set  $O_1(t) \cup O_2(t)$ .
  - 5: **(Local Search)** Apply local search operator given by Algorithm 4 on each individual in set  $O_1(t) \cup O_2(t)$ .
  - 6: **(Selection)** Select the best  $E$  individuals for the next population  $Pop(t+1)$  from set  $Pop(t) \cup O_1(t) \cup O_2(t)$ . Select the remaining  $Popsiz - E$  individuals for  $Pop(t+1)$  by roulette wheel selection also from set  $Pop(t) \cup O_1(t) \cup O_2(t)$ . Let  $t = t + 1$ .
  - 7: **(Stopping Criteria)** If a fixed number of generations reached, then stop and return the best individual  $\vec{I} = (\vec{\sigma}, \vec{A})$  in the current population; otherwise, go to Step 2.
- 



**Fig. 7** Flow chart of algorithm PAA-GA

**Table 2** Parameters of our HSCS

$P_i$	$g_i$	$w_i$	$e_i$	$f_i$	$r_i$	$o_i$
$P_1$	0.53	2.90	7.06	5.80	46.34	331.85
$P_2$	0.77	7.61	3.02	0.14	53.55	397.44
$P_3$	0.71	4.14	8.14	0.45	78.14	420.58
$P_4$	0.79	9.62	8.63	3.74	93.47	509.31
$P_5$	0.06	3.64	8.71	9.50	13.86	613.82
$P_6$	0.77	5.92	5.25	0.54	57.31	722.11
$P_7$	0.30	6.48	4.69	6.23	47.47	855.85
$P_8$	0.28	8.25	2.64	8.30	22.18	964.82
$P_9$	0.99	2.27	5.89	9.11	34.38	1175.66
$P_{10}$	0.98	5.34	6.95	2.44	17.06	1299.34
$P_{11}$	0.99	1.57	1.06	6.76	79.63	1474.28
$P_{12}$	0.10	7.99	5.75	1.03	31.81	1763.75
$P_{13}$	0.05	3.82	2.84	2.96	53.32	1768.40
$P_{14}$	0.95	4.01	3.01	9.80	17.40	1911.18
$P_{15}$	0.16	6.47	2.78	1.63	60.60	1943.74

the processing time obtained by PAA-GA agrees with that obtained by EA. If their processing times are in good agreement, then it surely proves that PAA-GA can obtain an OLP.

Assuming that the off-line times of all processors are infinite, thus we only take the processor release times into account. Note that EA requires a fixed load distribution sequence as its input in advance, so we set the OLDS obtained by PAA-GA as the input for algorithm EA. Table 3 records the comparison results for PAA-GA and EA. It can be observed from this table that PAA-GA obtains the same experimental results with EA for each test workload, including the same number of processors involved in workload computation and the same processing time. Therefore, we can make the conclusion that algorithm PAA-GA proposed in this chapter can obtain an OLP such that the processing time is minimized for divisible-load scheduling problems with processor release times.

Besides OLP, in order to prove that the proposed PAA-GA can also obtain an OLDS, we make a comparison between PAA-GA and EA with three different distribution sequences as its input, which are commonly used in previous studies: sequence in the order of increasing value of  $g_i$ , denoted as IG; sequence in the order of increasing value of  $w_i$ , denoted as IW; and sequence in the order of increasing value of release time  $r_i$ , denoted as IR.

Based on the parameters given in Table 2, sequences IG, IW, and IR are as follows,

**Table 3** Experimental results obtained by PAA-GA and EA with the same OLDS

$W_{total}$	Algorithm	$n$	$T$	$\vec{\sigma} = (\sigma_1, \sigma_2, \dots, \sigma_n)$
100	PAA-GA	12	110.615	(5, 8, 14, 12, 7, 13, 15, 1, 2, 11, 6, 3)
	EA	12	110.615	(5, 8, 14, 12, 7, 13, 15, 1, 2, 11, 6, 3)
200	PAA-GA	13	165.639	(5, 8, 12, 14, 13, 15, 7, 1, 11, 9, 2, 6, 3)
	EA	13	165.639	(5, 8, 12, 14, 13, 15, 7, 1, 11, 9, 2, 6, 3)
300	PAA-GA	14	216.182	(5, 8, 12, 7, 13, 15, 1, 3, 2, 11, 14, 9, 6, 10)
	EA	14	216.182	(5, 8, 12, 7, 13, 15, 1, 3, 2, 11, 14, 9, 6, 10)
400	PAA-GA	15	265.534	(5, 8, 12, 7, 13, 15, 1, 3, 2, 6, 11, 14, 9, 10, 4)
	EA	15	265.534	(5, 8, 12, 7, 13, 15, 1, 3, 2, 6, 11, 14, 9, 10, 4)
500	PAA-GA	15	314.943	(5, 8, 12, 7, 13, 15, 1, 3, 2, 6, 14, 11, 9, 10, 4)
	EA	15	314.943	(5, 8, 12, 7, 13, 15, 1, 3, 2, 6, 14, 11, 9, 10, 4)
600	PAA-GA	15	365.878	(5, 8, 12, 13, 15, 7, 1, 3, 2, 6, 14, 11, 9, 10, 4)
	EA	15	365.878	(5, 8, 12, 13, 15, 7, 1, 3, 2, 6, 14, 11, 9, 10, 4)
700	PAA-GA	15	414.793	(5, 8, 12, 13, 15, 7, 1, 3, 2, 6, 14, 11, 9, 10, 4)
	EA	15	414.793	(5, 8, 12, 13, 15, 7, 1, 3, 2, 6, 14, 11, 9, 10, 4)
800	PAA-GA	15	465.835	(5, 8, 12, 13, 15, 7, 1, 3, 2, 6, 14, 11, 9, 10, 4)
	EA	15	465.835	(5, 8, 12, 13, 15, 7, 1, 3, 2, 6, 14, 11, 9, 10, 4)
900	PAA-GA	15	516.877	(5, 8, 12, 13, 15, 7, 1, 3, 2, 6, 14, 11, 9, 10, 4)
	EA	15	516.877	(5, 8, 12, 13, 15, 7, 1, 3, 2, 6, 14, 11, 9, 10, 4)
1000	PAA-GA	15	567.761	(5, 12, 8, 13, 15, 7, 1, 3, 2, 6, 14, 11, 9, 10, 4)
	EA	15	567.761	(5, 12, 8, 13, 15, 7, 1, 3, 2, 6, 14, 11, 9, 10, 4)

$$IG = (13, 5, 12, 15, 8, 7, 1, 3, 2, 6, 4, 15, 10, 9, 11).$$

$$IW = (11, 9, 1, 5, 13, 14, 3, 10, 6, 15, 7, 2, 12, 8, 4).$$

$$IR = (5, 10, 14, 8, 12, 9, 1, 7, 13, 2, 6, 15, 3, 11, 4).$$

Table 4 records the experimental results obtained by algorithms PAA-GA, EA-IG, EA-IW, and EA-IR. It can be observed from Table 4 that for each test workload, the load distribution sequence obtained by PAA-GA is different from IG, IW, and IR. Moreover, for some test workloads, PAA-GA even obtains different numbers of processors involved in workload computation from EA with IG, IW, and IR, hence obtaining different load partition too. To be more intuitively, Fig. 8 illustrates the variation of processing time obtained by PAA-GA, EA-IG, EA-IW, and EA-IR along with different workload size. From this figure, we can observe that for each workload, the processing time obtained by the proposed PAA-GA is less than that by EA with three different load distribution sequences. The processing time obtained by PAA-GA shows a gain of about 10–25% compared to EA with IG and IR, and gained over 40% compared to EA with IW. Therefore, it is clear that PAA-GA outperforms over other strategies in achieving an optimal processing time, hence an efficient energy consumption too, for processor release time-aware divisible-load scheduling. Fur-

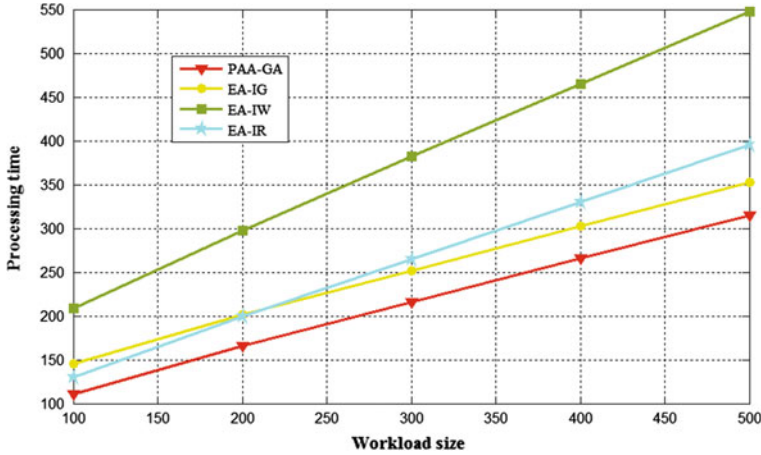
**Table 4** Experimental results obtained by PAA-GA and EA with three different load distribution sequences

$W_{total}$	Algorithm	$n$	$T$	$\vec{\sigma} = (\sigma_1, \sigma_2, \dots, \sigma_n)$
100	PAA-GA	12	110.615	(5, 8, 14, 12, 7, 13, 15, 1, 2, 11, 6, 3)
	EA-IG	11	145.418	(13, 5, 12, 15, 8, 7, 1, 3, 2, 6, 4)
	EA-IW	8	208.662	(11, 9, 1, 5, 13, 14, 3, 10)
	EA-IR	12	130.256	(5, 10, 14, 8, 12, 9, 1, 7, 13, 2, 6, 15)
200	PAA-GA	13	165.639	(5, 8, 12, 14, 13, 15, 7, 1, 11, 9, 2, 6, 3)
	EA-IG	15	201.189	(13, 5, 12, 15, 8, 7, 1, 3, 2, 6, 4, 14, 10, 9, 11)
	EA-IW	12	297.881	(11, 9, 1, 5, 13, 14, 3, 10, 6, 15, 7, 2)
	EA-IR	14	199.450	(5, 10, 14, 8, 12, 9, 1, 7, 13, 2, 6, 15, 3, 11)
300	PAA-GA	14	216.182	(5, 8, 12, 7, 13, 15, 1, 3, 2, 11, 14, 9, 6, 10)
	EA-IG	15	251.701	(13, 5, 12, 15, 8, 7, 1, 3, 2, 6, 4, 14, 10, 9, 11)
	EA-IW	14	382.499	(11, 9, 1, 5, 13, 14, 3, 10, 6, 15, 7, 2, 12, 8)
	EA-IR	15	265.008	(5, 10, 14, 8, 12, 9, 1, 7, 13, 2, 6, 15, 3, 11, 4)
400	PAA-GA	15	265.534	(5, 8, 12, 7, 13, 15, 1, 3, 2, 6, 11, 14, 9, 10, 4)
	EA-IG	15	302.213	(13, 5, 12, 15, 8, 7, 1, 3, 2, 6, 4, 14, 10, 9, 11)
	EA-IW	15	465.058	(11, 9, 1, 5, 13, 14, 3, 10, 6, 15, 7, 2, 12, 8, 4)
	EA-IR	15	330.451	(5, 10, 14, 8, 12, 9, 1, 7, 13, 2, 6, 15, 3, 11, 4)
500	PAA-GA	15	314.943	(5, 8, 12, 7, 13, 15, 1, 3, 2, 6, 14, 11, 9, 10, 4)
	EA-IG	15	352.725	(13, 5, 12, 15, 8, 7, 1, 3, 2, 6, 4, 14, 10, 9, 11)
	EA-IW	15	547.376	(11, 9, 1, 5, 13, 14, 3, 10, 6, 15, 7, 2, 12, 8, 4)
	EA-IR	15	395.894	(5, 10, 14, 8, 12, 9, 1, 7, 13, 2, 6, 15, 3, 11, 4)

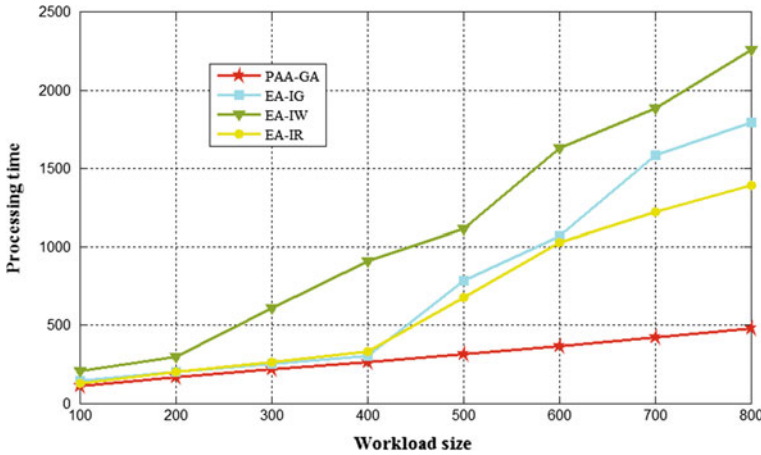
thermore, it can be seen from Fig. 8 that at first the processing time obtained by EA with IR is less than that by EA with IG, but as workload size increases, EA with IG outperforms EA with IR. This means that with increasing workload size, the influence of processor release times on the processing time becomes weaker, while the influence of load distribution sequence on processing time becomes stronger.

## 4.2 Evaluating the Performance of PAA-GA

By taking processor available time periods into account, we make a comparison between the proposed PAA-GA and EA with three commonly used load distribution sequences as its input: IG, IW, and IR. Given that the original EA proposed in [34] does not consider processor off-line times, those processors whose processing times exceed their off-line times need to be rescheduled. For simplicity, we reallocate those load fractions to the processor with the largest off-line time. Figure 9 shows the variation of processing time obtained by PAA-GA, EA-IG, EA-IW, and EA-IR along with different workload size. As shown in Fig. 9, for each workload,



**Fig. 8** Variation of processing time obtained by PAA-GA, EA-IG, EA-IW, and EA-IR along with different workload sizes for processor release-time aware divisible-load scheduling



**Fig. 9** Variation of processing time obtained by PAA-GA, EA-IG, EA-IW, and EA-IR along with different workload sizes for processor available-time aware divisible-load scheduling

the processing time obtained by PAA-GA is much less than that by EA with any of the load distribution sequences, and the time difference between them grows with increasing workload size. When workload size is as large as 800 in our experiment, the processing time obtained by PAA-GA shows a gain of 64% compared to EA with IR, gained about 72% compared to EA with IG, and gained over 77% compared to EA with IW. Therefore, it is clear that although the effect of available times has a greater influence on the performance as workload size increases, PAA-GA outperforms over other strategies as testified in our evaluations.

## 5 Conclusions

One of the key characteristics of sustainable computing systems is in efficiently managing available shared resources by designing judicious scheduling algorithms. By designing optimal, if not time-efficient scheduling algorithms, energy consumption is indirectly managed. Towards this effort, in this chapter, we have proposed an efficient processor availability-aware scheduling model to optimize the energy efficiency of heterogeneous sustainable computing systems. Using this model, we designed a genetic algorithm based global optimization strategy to derive an optimal load partition together with an optimal distribution sequence. This is an important contribution to the literature as this is the first time where the modeling is tuned to accommodate all influencing parameters (start-up overheads, processor availabilities, heterogeneous networks) to achieve a global optimal solution. We have conducted several experiments to demonstrate the correctness and effectiveness of the proposed algorithm PAA-GA. Experimental results showed that although the effect of processor available time periods has a greater influence on the performance as the workload size increases, the proposed PAA-GA reduced the processing time, hence the energy consumption too, by over 60% compared to other strategies. An important and an immediate useful extension to the study posed in this chapter is in developing a similar strategy for an arbitrary topology as real-life network based computing platforms seldom have regular topologies.

**Acknowledgements** This is a collaborative research work conducted jointly between Department of Electrical and Computer Engineering, National University of Singapore, Singapore, and School of Computer Science and Technology, Xidian University, China and supported by National Natural Science Foundation of China (No. 61402350, No. 61472297, and No. 61572391), the Fundamental Research Funds for the Central Universities (No. JB150307) and China Scholarship Council.

## References

1. L. Mashayekhy, M.M. Nejad, D. Grosu, Q. Zhang, W. Shi, Energy-aware scheduling of map-reduce jobs for big data applications. *IEEE Trans Parallel Distrib.* **26**, 2720–2733 (2015)
2. J. Cao, K. Li, I. Stojmenovic, Optimal power allocation and load distribution for multiple heterogeneous multicore server processors across clouds and data centers. *IEEE Trans. Comput.* **63**, 45–58 (2014)
3. Y.N. Xia, M.C. Zhou, X. Luo et al., A stochastic approach to analysis of energy-aware DVS-enabled cloud datacenters. *IEEE Trans. Syst. Man Cybern. A* **45**, 73–83 (2015)
4. X. Zhu, C. He, K. Li et al., Adaptive energy-efficient scheduling for real-time tasks on DVS-enabled heterogeneous clusters. *J. Parallel Distrib. Comput.* **72**, 751–763 (2012)
5. G. Terzopoulos, H. Karatza, Performance evaluation and energy consumption of a real-time heterogeneous grid system using DVS and DPM. *Simul. Model Pract. Theory* **36**, 33–43 (2013)
6. P. Zhou, W. Zheng, An efficient biobjective heuristic for scheduling workflows on heterogeneous DVS-enabled processors. *J. Appl. Math.* **2014** (2014)
7. B. Luo, S. Wang, W. Shi, Y. He, eCope: workload-aware elastic customization for power efficiency of high-end servers. *IEEE Trans. Cloud Comput.* **4**, 237–249 (2016)

8. P. Lama, Y. Guo, C. Jiang, X. Zhou, Autonomic performance and power control for co-located web applications in virtualized datacenters. *IEEE Trans. Parallel Distrib.* **27**, 1289–1302 (2016)
9. Z. Xiao, W. Song, Q. Chen, Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE Trans. Parallel Distrib.* **24**, 1107–1117 (2013)
10. U. Wajid, P. Plebani, B. Pernici et al., On achieving energy efficiency and reducing CO<sub>2</sub> footprint in cloud computing. *IEEE Trans. Cloud Comput.* **4**, 138–151 (2016)
11. H. Liu, H. Jin, C.Z. Xu et al., Performance and energy modeling for live migration of virtual machines. *Cluster Comput.* **16**, 249–264 (2013)
12. A. Corradi, M. Fanelli, L. Foschini, VM consolidation: a real case based on OpenStack Cloud. *Future Gener. Comput. Syst.* **32**, 118–127 (2014)
13. X. Dai, M. Wang, B. Benasou, On achieving energy efficiency and reducing CO<sub>2</sub> footprint in cloud computing. *IEEE Trans. Cloud Comput.* **4**, 210–221 (2016)
14. E. Feller, L. Ramakrishnan, C. Morin, Performance and energy efficiency of big data applications in cloud environments: a Hadoop case study. *J. Parallel Distrib. Comput.* **79**, 80–89 (2015)
15. T.G. Robertazzi, Ten reasons to use divisible load theory. *Computer* **36**, 63–68 (2003)
16. S. Momcilovic, A. Ilic, N. Roma, L. Sousa, Dynamic load balancing for real-time video encoding on heterogeneous CPU+GPU systems. *IEEE Trans. Multimedia* **16**, 108–121 (2014)
17. A. Ilic, S. Momcilovic, N. Roma, L. Sousa, Adaptive scheduling framework for real-time video encoding on heterogeneous systems. *IEEE Trans. Circ. Syst. Video Technol.* **26**, 597–611 (2016)
18. S. Suresh, H. Huang, H.J. Kim, Scheduling in compute cloud with multiple data banks using divisible load paradigm. *IEEE Trans. Aerosp. Electron. Syst.* **51**, 1288–1297 (2015)
19. Z. Ying, T.G. Robertazzi, Signature searching in a networked collection of files. *IEEE Trans. Parallel Distrib.* **25**, 1339–1348 (2014)
20. V. Mani, D. Ghose, Distributed computation in linear networks: closed-form solutions. *IEEE Trans. Aerosp. Electron. Syst.* **30**, 471–483 (1994)
21. T.E. Carroll, D. Grosu, Strategyproof mechanisms for scheduling divisible loads in bus-networked distributed systems. *IEEE Trans. Parallel Distrib.* **19**, 1124–1135 (2008)
22. S. Ghanbari, M. Othman, M.R.A. Bakar, W.J. Leong, Multi-objective method for divisible load scheduling in multi-level tree network. *Future Gener. Comput. Syst.* **54**, 132–143 (2016)
23. Z. Zhang, T.G. Robertazzi, Scheduling divisible loads in Gaussian, mesh and torus network of processors. *IEEE Trans. Comput.* **64**, 3249–3264 (2015)
24. C.Y. Chen, C.P. Chu, Novel methods for divisible load distribution with start-up costs on a complete b-Ary tree. *IEEE Trans. Parallel Distrib.* **26**, 2836–2848 (2015)
25. K. Wang, T.G. Robertazzi, Scheduling divisible loads with nonlinear communication time. *IEEE Trans. Aerosp. Electron. Syst.* **51**, 2479–2485 (2015)
26. V. Bharadwaj, D. Ghose, V. Mani, Optimal sequencing and arrangement in distributed single-level tree networks with communication delays. *IEEE Trans. Parallel Distrib.* **5**, 968–976 (1994)
27. H.J. Kim, G.I. Jee, J.G. Lee, Optimal load distribution for tree network processors. *IEEE Trans. Aerosp. Electron. Syst.* **32**, 607–612 (1996)
28. B. Veeravalli, X. Li, C.C. Ko, On the influence of start-up costs in scheduling divisible loads on bus networks. *IEEE Trans. Parallel Distrib.* **11**, 1288–1305 (2000)
29. S. Mingsheng, Optimal algorithm for scheduling large divisible workload on heterogeneous system. *Appl. Math. Model.* **32**, 1682–1695 (2008)
30. V. Bharadwaj, G. Barlas, Scheduling divisible loads with processor release times and finite size buffer capacity constraints in bus networks. *Cluster Comput.* **6**, 63–74 (2003)
31. M. Gallet, Y. Robert, F. Vivien, Comments on “design and performance evaluation of load distribution strategies for multiple loads on heterogeneous linear daisy chain networks”. *J. Parallel Distrib. Comput.* **68**, 1021–1031 (2008)
32. M. Hu, B. Veeravalli, Requirement-aware strategies with arbitrary processor release times for scheduling multiple divisible loads. *IEEE Trans. Parallel Distrib.* **22**, 1697–1704 (2011)



33. S. Suresh, V. Mani, S.N. Omkar, H.J. Kim, A real coded genetic algorithm for data partitioning and scheduling in networks with arbitrary processor release time, in *Asia-Pacific Computer Systems Architecture Conference* (2005), pp. 529–539
34. K. Choi, T.G. Robertazzi, An exhaustive approach to release time aware divisible load scheduling. *Int. J. Internet Distrib. Comput. Syst.* **1**, 40–50 (2011)
35. H.J. Kim, A novel optimal load distribution algorithm for divisible loads. *Cluster Comput.* **6**, 41–46 (2003)
36. D.C. Snowdon, S.M. Petters, G. Heiser, Accurate on-line prediction of processor and memory energy usage under voltage scaling, in *7th ACM and IEEE International Conference on Embedded Software* (2007), pp. 84–93
37. K. Li, X. Tang, K. Li, Energy-efficient stochastic task scheduling on heterogeneous computing systems. *IEEE Trans. Parall. Distrib.* **25**, 2867–2876 (2014)
38. D.M. Bui, H.Q. Nguyen, Y. Yoon, S. Jun, M.B. Amin, S. Lee, Gaussian process for predicting CPU utilization and its application to energy efficiency. *Appl. Intell.* **43**, 874–891 (2015)
39. J.H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. U. Michigan Press. (1975)
40. L. Hernando, A. Mendiburu, J.A. Lozano, A tunable generator of instances of permutation-based combinatorial optimization problems. *IEEE T. Evolut. Comput.* **20**, 165–179 (2016)
41. B.W. Goldman, W.F. Punch, Analysis of Cartesian genetic programming’s evolutionary mechanisms. *IEEE Trans. Evolut. Comput.* **19**, 359–373 (2015)

Intelligent Decision Support Systems for Sustainable  
Computing

Paradigms and Applications

Sangaiah, A.K.; Abraham, A.; Siarry, P.; Sheng, M. (Eds.)

2017, XVI, 289 p. 108 illus., 86 illus. in color., Hardcover

ISBN: 978-3-319-53152-6