J Supercomput (2014) 68:1347–1377
DOI 10.1007/s11227-014-1090-4

A resource-aware scheduling algorithm with reduced task duplication on heterogeneous computing systems

Jing Mei · Kenli Li · Keqin Li

Published online: 29 January 2014

Abstract To satisfy the high-performance requirements of application executions, many kinds of task scheduling algorithms have been proposed. Among them, duplication-based scheduling algorithms achieve higher performance compared to others. However, because of their greedy feature, they duplicate parents of each task as long as the finish time can be reduced, which leads to a superfluous consumption of resource. However, a large amount of duplications are unnecessary because slight delay of some uncritical tasks does not affect the overall makespan. Moreover, these redundant duplications would occupy the resources, delay the execution of subsequent tasks, and increase the schedule makespan consequently. In this paper, we propose a novel duplication-based algorithm designed to overcome the above drawbacks. The proposed algorithm is to schedule tasks with the least redundant duplications. An optimizing scheme is introduced to search and remove redundancy for a schedule generated by the proposed algorithm further. Randomly generated directed acyclic graphs and two real-world applications are tested in our experiments. Experimental

J. Mei · Kenli Li (✉) · Keqin Li
College of Information Science and Engineering, Hunan University,
Changsha 410082, Hunan, China
e-mail: lkl@hnu.edu.cn

J. Mei
e-mail: jingmei1988@163.com

Kenli Li
National Supercomputing Center in Changsha,
Changsha 410082, Hunan, China

Keqin Li
Department of Computer Science, State University of New York,
New Paltz, NY 12561, USA
e-mail: lik@newpaltz.edu

results show that the proposed algorithm can save up to 15.59 % resource consumption compared with the other algorithms. The makespan has improvement as well.

Keywords Directed acyclic graph · Duplication-based task scheduling · Heterogeneous computing system · Resource awareness

1 Introduction

In the past decade, to satisfy the high-performance requirements of application executions, much attention has been focused on the task scheduling problem for applications on heterogeneous computing systems. A *heterogeneous computing* (HC) system is defined as a distributed suite of computing machines with different capabilities interconnected by different high-speed links utilized to execute parallel applications [1,2]. An application is represented in the form of a *directed acyclic graph* (DAG) consisting of many tasks. Task computation cost and intertask communication cost in a DAG are determined for an HC system via estimation and benchmarking techniques [3–5].

The challenge of task scheduling is to find an assignment of the tasks of an application onto the processors of a target HC system, which results in minimal schedule length, while respecting the precedence constraints among tasks [6]. Finding a schedule of the minimal length for a given task graph is, in its general form, an NP-hard problem [7,8]. Hence, many heuristics are proposed to obtain sub-optimal scheduling solutions. Task scheduling has been extensively studied and various heuristics have been proposed in the literature [9–17]. The general task scheduling algorithms can be classified into a variety of categories, such as list scheduling algorithms, clustering algorithms, duplication-based algorithms, and so on. The objective of existing algorithms is to achieve a minimal schedule length. To achieve this goal, they always scarify a large amount of resource, and hence there is a sharp increase in energy consumption. In recent years, with the growing advocacy for green computing systems, energy conservation has become an important issue and has gained particular interest. In this paper, we present a new task scheduling algorithm on HC systems which considers resource consumption as well as makespan.

Among all algorithms, the kind of duplication-based algorithms have the best performance in terms of makespan. The idea of duplication-based algorithms is to schedule a task graph by mapping some of its tasks to several processors, which reduces communication among tasks. The quality of a solution generated by a duplication-based algorithm is usually much better than that generated by a nonduplication-based algorithm in terms of makespan. A duplication-based algorithm belongs to the class of greedy algorithms, as each task is assigned a processor which allows the earliest finish time of the task, and a parent is duplicated for a task as long as its finish time can be reduced. Due to the greedy mechanism, some of the tasks of an application are executed repeatedly, which leads to superfluous consumption of resource. According to our analysis, a large amount of duplications are redundant. Slight delay of some uncritical tasks without those redundant duplications does not affect the overall makespan. These redundant copies not only waste a huge amount of processor resource, but also occupy the locations of subsequent tasks, hence delaying the overall makespan.

Therefore, in this paper, we attempt to explore the method of reducing redundant duplications during the scheduling process to overcome the above-mentioned drawback. In addition, a further optimizing scheme is proposed for a schedule generated by our algorithm.

In our paper, all information of a DAG including execution times of tasks, the data sizes of communication between tasks, and task dependencies are known *a priori*, which are necessary information required in static scheduling. Static task scheduling takes place during compile time before task execution. Once a schedule is determined, tasks can be executed following the order and processor assignments.

The contributions of this paper are summarized as follows.

- We propose a novel resource-aware scheduling algorithm called RADS, which searches and deletes redundant task duplications dynamically in the process of scheduling.
- A further optimizing scheme is designed for the schedules generated by our algorithm, which can further reduce resource consumption without degrading the makespan.
- Experiments are conducted to verify that both the proposed algorithm and the optimizing scheme can achieve good performance in terms of makespan and resource efficiency. The factors affecting the performance of our algorithm are analyzed.

The remainder of this paper is organized as follows. Section 2 reviews some related work, including some typical heuristic algorithms on HC systems and the current research status of resource-aware algorithms. In Sect. 3, we define the scheduling problem and present related models. Our proposed RADS algorithm is developed in detail in Sect. 4, together with its time complexity analysis. In addition, an example is also provided in this section to explain our algorithm better. Section 5 describes the optimizing scheme of RADS. The experimental results are demonstrated in Sect. 6, together with an analysis of the impacts of different parameters on the performance of the proposed method. Finally, we conclude this paper and give an overview of future work in Sect. 7.

2 Related work

DAG scheduling algorithms can be classified into two categories with respect to whether to duplicate tasks or not. List scheduling and clustering algorithms are two kinds of nonduplication-based algorithms. List scheduling algorithms provide schedules of good quality and their performance is comparable with other categories at lower time complexities. Due to no duplication, list scheduling algorithms consume less processor resource than duplication-based algorithms. However, the high communication cost among tasks limits the performance of list scheduling in terms of makespan. Classical examples of list scheduling algorithms are dynamic critical path (DCP) [18], heterogeneous earliest finish time (HEFT) [13], critical path on a processor (CPOP) [13], and the longest dynamic critical path (LDCP) [11]. Clustering algorithms merge tasks in a DAG to an unlimited number of clusters, and tasks in a cluster are scheduled on the same processor. Some examples in this category include clustering for heterogeneous processors (CHP) [19], clustering and scheduling system (CASS) [20], objective-flexible clustering algorithm (OFCA) [21], and so on. Duplication-based algorithms achieve much better performance in terms of makespan compared with nonduplication-based algorithms by mapping some tasks redundantly, which reduces intertask communication. Many duplication-based algorithms are proposed in recent years, for example, selective duplication (SD) [17], heterogeneous limited duplication (HLD) [12], heterogeneous critical parents with fast duplicator (HCPFD) [15], heterogeneous earliest finish with duplication (HEFD) [22], and so on. As mentioned before, duplication-based algorithms improve the makespan at the cost of significant waste of processor resource. There are some researches such as [23] combining DVS technique with duplication strategy to reduce energy consumption and increase processor utilization. The resource wasting problem still exists due to duplication.

To overcome the above shortcoming of duplication-based algorithms, two novel algorithms to solve the problem of resource waste were proposed in [24] and [25]. The algorithm proposed in [24] is designed to delete the original copies of join nodes when some particular conditions are satisfied. The algorithm proposed in [25] consists of two sub-algorithms, namely, SDS sub-algorithm, which assigns tasks to processors, and SC sub-algorithm, which merges two partial schedules to one. These two algorithms can reduce duplications efficiently, but they aim at the scheduling problem on homogeneous systems. Therefore, they are not applicable to DAG scheduling on HC systems.

Our research is different from all existing works, because the focus of our work is on resource-aware scheduling of applications on HC systems, and the aim is at reducing resource consumption as well as makespan, whereas existing works do not take resource efficiency into account or are based on homogeneous systems. In our previous work [26], we proposed an initial algorithm to delete redundant copies of tasks. In this paper, we conduct further research on this issue.

3 Models

A task scheduling system model consists of a target computing platform and an application model. In this section, we present our computing system model and application model used in this paper. Moreover, the performance measures used to evaluate the performance of the scheduling algorithms are introduced as well. In Table 1, we summarize all the notations used in the paper to improve the readability.

3.1 Computing system model

This paper studies the task scheduling problem for applications on heterogeneous computing systems. Let $P = \{p_i \mid 0 \leq i \leq m - 1\}$ be a set of m processors with different capacities. The capacity of a processor in processing a task depends on how well the processor's architecture matches the task's processing requirement. A task scheduled on a better-suited processor will take shorter execution time than on a less-suited processor. The best processor for one task may be the worst one for another task. This HC model is described by [27] and used in [9, 13–15].

Table 1 Notations used in this paper

Notation	Description
P	The set of m processors in system, $P = \{p_i \| 0 \leq i \leq m-1\}$
G	A DAG representing an application
V	The set of n nodes in a DAG, $V = \{v_i \| 1 \leq i \leq n\}$
w	The computation cost of tasks on processors, $w_{i,j} \in w$ representing the computation cost of v_i on p_j
c	The communication cost between tasks, $c(e_{ij}) \in c$ representing the communication cost between v_i and v_j
$\overline{w_i}$	The average cost of task v_i
$pare_i(v_i)$	The immediate parent set of v_i
$pare_m(v_i)$	The mediate parent set of v_i
$child_i(v_i)$	The immediate parent set of v_i
$child_m(v_i)$	The mediate parent set of v_i
S	a schedule
$st(v_i, p_k)$	The start time of v_i on p_k
$ft(v_i, p_k)$	The finish time of v_i on p_k
$Pbt(S)$	The processor busy time of schedule S
$\pi(v_i)$	The processor set where have a copy of v_i
$rank_u(v_i)$	The upward rank of v_i
$est(v_i, p_k)$	The earliest start time of v_i on p_k
$eft(v_i, p_k)$	The earliest finish time of v_i on p_k
$dat(v_i, p_k)$	The time that the data of v_i arrives at p_k
$lft(v_i, p_k)$	The latest finish time of v_i on p_k
$M_{i,k}$	The important immediate parent of v_i on p_k
H	the idle slot set of processor, which consists of a series of idle slot $< h_r^s, h_r^f >$
$< v_i, p_k >$	a copy of v_i on p_k
$\xi(v_i)$	the copy of v_i with the earliest finish time
$\vartheta(v_i)$	the processor of $\xi(v_i)$
$lft(v_i, p_k)$	the latest finish time of v_i on p_k
$V_lc(v_i, p_k)$	the local children set of (v_i, p_k)
$V_oc(v_i, p_k)$	the children copy set of v_i without local duplication of v_i
$V_lp(v_i, p_k)$	the local parent set of (v_i, p_k)
$V_op(v_i, p_k)$	the fixed copies of the off-processor parents of (v_i, p_k)

3.2 Application model

An application is represented by a *directed acyclic graph* (DAG) $G(V, E, w, c)$, which consists of a set of nodes $V = \{v_i \mid 1 \leq i \leq n\}$ representing the tasks of the application, and a set of directed edges E representing dependencies among tasks. A positive weight $w_{i,k} \in w$ represents the computation time of task v_i on processor p_k for $1 \leq i \leq n$ and $0 \leq k \leq m-1$. A nonnegative weight $c(e_{ij})$ associated with an edge $e_{ij} \in E$ represents the communication time to send data from v_i to v_j.

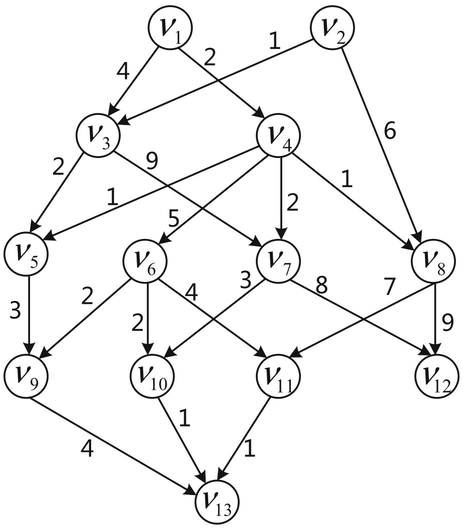


Fig. 1 A simple DAG representing an application graph with precedence constraints

Table 2 Computation costs of tasks in Fig. 1

Task node	p_0	p_1	p_2	p_3	$\overline{w_i}$
v_1	3	2	1	2	2
v_2	1	3	1	3	2
v_3	2	1	3	2	2
v_4	2	3	4	1	2.5
v_5	1	1	1	1	1
v_6	5	8	7	4	6
v_7	2	1	2	3	2
v_8	4	3	2	3	3
v_9	1	3	2	2	2
v_{10}	6	3	4	7	5
v_{11}	2	1	3	2	2
v_{12}	2	3	4	2	2.75
v_{13}	2	1	2	3	2

We demonstrate a simple DAG in Fig. 1 which consists of 13 nodes. Table 2 lists the computation times. In Table 2, the columns of p_0 to p_3 represent the computation cost of tasks on different processors. The average computation cost of task v_i is defined as $\overline{w_i} = \frac{1}{n}\sum_{k=1}^{n} w_{i,k}$, which is the average of the columns. The sample graph will be employed as an example throughout the following sections.

An edge $e_{ij} \in E$ from node v_i to v_j, where $v_i, v_j \in V$, represents v_j's receiving data from v_i, where v_i is called an *immediate parent* of v_j, and v_j is an *immediate child* of v_i. The immediate parent set of task v_i is denoted by $pare_i(v_i)$, and the immediate child set of task v_i is denoted by $child_i(v_i)$. For example, the immediate parent set and child set of task v_7 are $\{v_3, v_4\}$ and $\{v_{10}, v_{12}\}$, respectively. In a DAG, if $e_{ij} \in E$, and $e_{jk} \in E$, we term v_i a *mediate parent* of v_k, and v_k a *mediate child* of v_i. The mediate child set and mediate parent set of task v_i are denoted by $child_m(v_i)$ and $pare_m(v_i)$, respectively. For example, the mediate parent set and child set of task v_7 are $\{v_1, v_2\}$ and $\{v_{13}\}$, respectively.

A task having no parent is called an *entry task*, such as tasks v_1 and v_2 in Fig. 1. A task having no child is called an *exit task*, such as tasks v_{12} and v_{13}. A DAG may have multiple entry tasks and multiple exit tasks.

3.3 Performance measures

In this paper, we adopt two important measures to evaluate the performance of scheduling algorithms. Because the original objective of task scheduling is the fastest execution of an application, the schedule length, or makespan, is undoubtedly one of the most important criteria.

For a task v_i scheduled on processor p_k, let $st(v_i, p_k)$ and $ft(v_i, p_k)$ represent its *start time* and *finish time*. Because preemptive execution is not allowed, $ft(v_i, p_k) = st(v_i, p_k) + w_{i,k}$. The makespan is defined as

$$\text{makespan} = \max\{ft(v_i, p_k) \mid v_i \text{ is an exit task}\}. \tag{1}$$

Due to resource awareness of our algorithm, the processor resource consumed by a schedule has to be measured, and the criterion is processor busy time (Pbt). The processor busy time is the total period of time when processors execute tasks. It measures the processor requirement of a schedule. Let $S = \{(v_i, p_k, st(v_i, p_k), ft(v_i, p_k)) \mid v_i \in V, p_k \in P\}$ be a schedule generated by any algorithm, then its processor busy time is calculated by

$$Pbt(S) = \sum_{v_i \in V} \sum_{p_k \in \pi(v_i)} (ft(v_i, p_k) - st(v_i, p_k)), \tag{2}$$

where $\pi(v_i)$ is the set of processors which have a copy of v_i.

Figure 2 presents a duplication-based schedule of the DAG in Fig. 1, whose processor busy time is 33.

4 Proposed algorithm

In the past decade, many duplication-based algorithms for heterogeneous computing environments, such as HLD [12], HCPFD [15], and HEFD [22], have been proposed. These algorithms achieved good performance in terms of makespan. However, they do not take resource consumption into consideration. In this section, we present a *resource-aware scheduling algorithm with duplications* (RADS), which considers resource efficiency as well as makespan. The detailed description of our algorithm is presented in the following subsections.

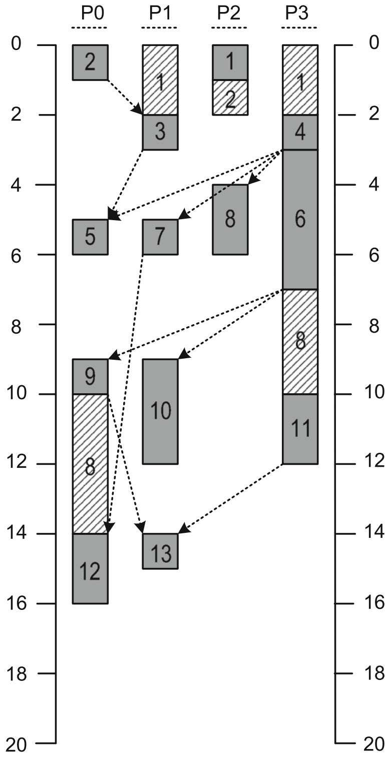


Fig. 2 A schedule of DAG in Fig. 1

4.1 Task priority

In RADS, all tasks in a DAG are assigned with scheduling priorities based on upward ranking [13]. The task with the highest priority is scheduled first. The upward rank of task v_i is recursively calculated by

$$rank_u(v_i) = \overline{w_i} + \max_{v_j \in child_i(v_i)} (c(e_{ij}) + rank_u(v_j)), \tag{3}$$

where $child_i(v_i)$ is the set of immediate children of task v_i. The rank value of exit task v_{exit} is

$$rank_{\mathrm{u}}(v_{exit}) = \overline{w_{exit}}. \tag{4}$$

The upward ranks of all tasks in the example DAG are listed in Table 3. From the example we can see that the rank value of a task must be greater than that of its children, that is, a parent must be scheduled before a child.

4.2 RADS algorithm

In this subsection, we give a detailed description on our proposed RADS algorithm. The tasks are scheduled in nonincreasing order of $rank_u$. The scheduling process of

Table 3 The upward ranks of tasks

Task node	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}
$\mathrm{rank_u}$	30	27	24	23.5	12	16	13	15	8	8	5	2.75	2

Algorithm 1 RADS algorithm

```
Require: The application $V$ and the processor set $P$
Ensure: The generated schedule $S$
Construct a task priority queue $\Gamma$ in a nonincreasing order of $rank_u$
$\Theta \leftarrow \emptyset$ \\ $\Theta$ is the task set whose redundant copies have been deleted
$C \leftarrow \emptyset$ \\ $C$ is the task set in which all children have been scheduled in current loop
while $\Gamma$ is not empty do
  $v_i \leftarrow$ first unscheduled task in $\Gamma$
  for all $p_k \in P$ do
    call cal_eft($v_i, p_k$) and record $eft(v_i, p_k)$
  end for
  schedule $v_i$ on $p_k$ with minimum $eft(v_i, p_k)$
  $child(v_j) \leftarrow \{child(v_j) - v_i\} \; \forall v_j \in V$
  $flag \leftarrow false$
  for all task $v_j$ in $V$ do
    if $child(v_j) = \emptyset$ and $v_j \notin \Theta$ then
      $C \leftarrow C \cup v_j$
      $flag \leftarrow true$
    end if
  end for
  if $flag = true$ then
    call Algorithm 3 to delete redundant copies
    $\Gamma \leftarrow \Gamma \cup C$
    $C \leftarrow \emptyset$
  end if
end while
```

each task is divided into two stages, namely, the task-mapping stage, in which the task is mapped to the processor which results in the earliest finish time of the task, and the redundancy deletion stage, in which redundant copies of the task are found out and deleted from the schedule. The pseudo-code of the RADS algorithm is given in Algorithms 1–3.

4.2.1 Task mapping stage

In the task-mapping stage, the *earliest finish time* (eft) of each task v_i is calculated for each processor $p_k \in P$, which is denoted by $eft(v_i, p_k)$. Task v_i is mapped to the processor that provides the minimum eft. Notice that $eft(v_i, p_k)$ is the earliest finish time of v_i on p_k. To calculate $eft(v_i, p_k)$, all parents of v_i must have been executed before v_i and their finish times must be known *in priori*. Let v_ϱ be a parent of v_i which is assigned to p_ε, and its finish time is denoted by $ft(v_\varrho, p_\varepsilon)$. Because task v_ϱ may be assigned to multiple processors due to duplication, v_i receives data from the one whose data arrive earliest. Hence, the *data arrival time* (dat) of v_ϱ on processor p_k is calculated by the following equation,

Algorithm 2 cal_eft(v_i, p_k)

```
calculate $eft^{'}(v_i, p_k)$ without duplication
$M_{i,k} \leftarrow$ the $miip$ of task $v_i$ on $p_k$
**if** $M_{i,k}$ does not exist or is already scheduled on $p_k$ **then**
  return $eft^{'}(v_i, p_k)$
**else**
  **if** a suitable slot exists for $M_{i,k}$ on $p_k$ **then**
    duplicate $M_{i,k}$ and calculate $eft^{''}(v_i, p_k)$
  **end if**
  **if** $eft^{''}(v_i, p_k) < eft^{'}(v_i, p_k)$ **then**
    return $eft^{''}(v_i, p_k)$
  **else**
    return $eft^{'}(v_i, p_k)$
  **end if**
**end if**
```

$$dat(v_\varrho, p_k) = \min_{p_\varepsilon \in \pi(v_\varrho)} \{ft(v_\varrho, p_\varepsilon) + \overline{c(e_{\varrho,i})}\}, \tag{5}$$

where $\pi(v_\varrho)$ is the set of processors which have a copy of v_ϱ and $\overline{c(e_{\varrho,i})}$ is the communication cost between v_ϱ and v_i. It is clear that $\overline{c(e_{\varrho,i})} = c(e_{\varrho,i})$ if $p_k \neq p_\varepsilon$, and $\overline{c(e_{\varrho,i})} = 0$ otherwise.

Once all parents of v_i have been scheduled, the earliest finish time of task v_i on processor p_k can be calculated by

$$eft(v_i, p_k) = \max_{v_\varrho \in pare_i(v_i)} \{dat(v_\varrho, p_k)\} + w_{i,k}, \tag{6}$$

where $pare_i(v_i)$ is the immediate parent set of v_i.

According to Eq. (6), we can conclude that $eft(v_i, p_k)$ is mainly determined by the *most important immediate parent* ($miip$) that has the latest data arrival time. Therefore, reducing the dat of $miip$ can minimize the eft of task v_i. RADS adopts duplication strategy to realize the goal, and the detailed process is shown in Algorithm 2. First, RADS calculates the earliest finish time of v_i on p_k without duplication, which is denoted by $eft^{'}(v_i, p_k)$. Second, RADS finds out the $miip$ of v_i on p_k, which is denoted by $M_{i,k}$, and recalculates the finish time of v_i, $eft^{''}(v_i, p_k)$, assuming that $M_{i,k}$ is duplicated on the same processor p_k. Third, by comparing the two results, RADS selects and records the schedule which results in a smaller eft.

To duplicate the $miip$ of task v_i on processor p_k, where v_j, a suitable scheduling hole should be exploited. We assume that H is the free slot set on processor p_k, which consists of a series of free slots $< h_r^s, h_r^f >$. A suitable scheduling hole to duplicate v_j must satisfy the following equation.

$$\max\{dat(v_j, p_k), h_r^s\} + w_{j,i} \leq h_r^f \tag{7}$$

In all the suitable scheduling holes which satisfy Eq. (7), we select the earliest one to duplicate v_j to minimize the eft of v_i as much as possible.

After all processors $p_k \in P$ are traversed, eft values of v_i on all $p_k \in P$ are calculated. RADS assigns task v_i to the processor p_k with minimum $eft\ (v_i, p_k)$. Till now, the current schedule of v_i has been generated.

According to our algorithm, we can see that a task may have multiple copies in a schedule, and all of them could provide data for their children. However, a child chooses to receive data from the parent copy whose data arrive the earliest. In general, if there exists a parent copy on the same processor as a child, the child receives data from the parent copy with a higher priority; otherwise, it receives data from a parent copy assigned to a different processor. To distinguish the two kinds of parent copies, we give a definition as follows.

Definition 1 Let v_c be a child of v_i, and (v_c, p_l) represents a copy of v_c on p_l. If there exists a copy of v_i on the same processor p_l, denoted by (v_i, p_l), such that (v_i, p_l) is scheduled earlier than (v_c, p_l), and (v_c, p_l) is to receive data from (v_i, p_l) without communication, then (v_i, p_l) is called a *local parent* of v_c and (v_c, p_l) is called a *local child* of v_i. If (v_c, p_l) does not have a local parent and it is required to receive data from another copy of v_i on processor p_k, then the copy (v_i, p_k) is called an *off-processor parent* of (v_c, p_l) and (v_c, p_l) is called an *off-processor child* of v_i.

Definition 2 The *parent set* and *child set* of v_i, represented by $parent(v_i)$ and $child(v_i)$, are $parent(v_i) = pare_i(v_i) \cup pare_m(v_i)$, and $child(v_i) = child_i(v_i) \cup child_m(v_i)$, respectively.

After the current task v_i has been scheduled, v_i is removed from the child set of all tasks in line 10. If there exists a task whose child set becomes empty, RADS turns to the second stage to search and delete the redundant copies.

4.2.2 Redundancy deletion stage

In the task-mapping stage, a task is assigned to the processor which results in the minimal finish time. To execute tasks as early as possible, RADS tries to duplicate the $miip$ for each task. Hence, the task-mapping stage has the greedy feature. Due to massive duplication, a task may be assigned onto more than one processor, hence it is possible to generate many redundant copies. An example scenario is given to demonstrate how redundant copies are generated.

Let task v_i be mapped to p_k firstly. Assume that all children of v_i are mapped to processors different from p_k, and each child has a duplicated copy of v_i. Then, the original copy of v_i on p_k becomes a redundant one. Deleting the redundant copy from a schedule does not affect the overall makespan. The redundancy deletion phase aims at discriminating and deleting this kind of redundant copies. The deletion action has two objectives. The first one is to decrease resource consumption and the other is to release resource for subsequent tasks.

In scheduling, it is very critical when and how to judge if a task copy is redundant. In our algorithm, we judge it when all children of a task are scheduled. That is because it is concluded by analyzing that at that instant the redundant copies are really redundancies and deleting them will not affect the whole performance in terms of makespan. Let v_i be a task of a given application. For its immediate children in $child_i(v_i)$, their finish times depend on the finish time of v_i apparently. For the mediate children in $child_m(v_i)$, if their finish times rely on duplicated parents, and these duplicated parents in $child_i(v_i)$ depend on the execution of v_i, the tasks in $child_m(v_i)$ are affected by the finish time of v_i. Therefore, to not deteriorate the performance, we discriminate a task only after all its children, including immediate and mediate ones, have been scheduled. This process is described in lines 10–16. After a task v_i is scheduled, it is deleted from the child sets of all tasks in line 1. And then, all tasks are traversed to find out the tasks whose child set becomes empty in this loop 12–16. If the task set C is not empty, Algorithm 3 is called to find out the redundancies.

Another important issue considered in the redundancy deletion stage is how to determine a copy to be redundant. A copy of a task can be deleted only when the other copies of this task can provide data needed by all its children. We give some definitions as follows.

Definition 3 In a schedule, the *fixed copy* of task v_i, denoted by $\xi(v_i)$, is defined as the one with the earliest finish time, and the corresponding processor is called its *fixed processor*, denoted by $\vartheta(v_i)$.

Because the fixed copy of a task is the copy with the earliest finish time among all copies, it is the default copy which provides data for those children without local parent. Since the fixed copy can satisfy all off-processor children dependencies, the only purpose of other copies is to provide data for their local children.

Lemma 1 *Let task v_ϱ be a parent of task v_i. v_ϱ and v_i have copies on processors p_ε and p_k separately. We say that the copy $(v_\varrho, p_\varepsilon)$ can provide data for (v_i, p_k) if it satisfies the following condition,*

$$ft(v_\varrho, p_\varepsilon) + \overline{c(e_{\varrho,i})} \leq st(v_i, p_k), \tag{8}$$

where $\overline{c(e_{\varrho,i})} = c(e_{\varrho,i})$ if $p_i = p_j$, and $\overline{c(e_{\varrho,i})} = 0$ otherwise.

Algorithm 3 Redundancy deletion stage

```
Require: The set of tasks $C$ whose redundant copies are to be deleted
Ensure: The schedule $S$ after removing the redundant copies
for all tasks $v_j \in C$ in nondecreasing order of $rank$ do
  if $v_j$ has multiple copies then
    for each processor $p_k$ that has a copy of $v_j$ do
      delete the current copy of $v_j$, denoted by $(v_j, p_k)$
      if $(v_j, p_k)$ is the fixed copy then
        determine the new $\xi(v_j)$
      end if
      if $\xi(v_j)$ cannot afford the data of $(v_j, p_k)$'s local children then
        undo the deleting operation of $(v_j, p_k)$
      end if
    end for
    update $\xi(v_j)$
  end if
end for
```

Let (v_i, p_k) represent a copy of v_i on p_k. The steps of deciding if (v_i, p_k) is a redundant copy are described as follows. First, (v_i, p_k) is deleted from the schedule. If (v_i, p_k) is the fixed copy of v_i, the duplicated copy of v_i with the second earliest finish time is selected as the new fixed copy. Then, we decide whether the new fixed copy can provide data for the children without local parent of v_i. If (v_i, p_k) is a duplicated copy, we decide if the fixed copy of v_i can provide data for (v_i, p_k)'s local children. If the constraints between tasks can still be satisfied, we can determine (v_i, p_k) to be redundant; otherwise, (v_i, p_k) is not redundant and cannot be deleted.

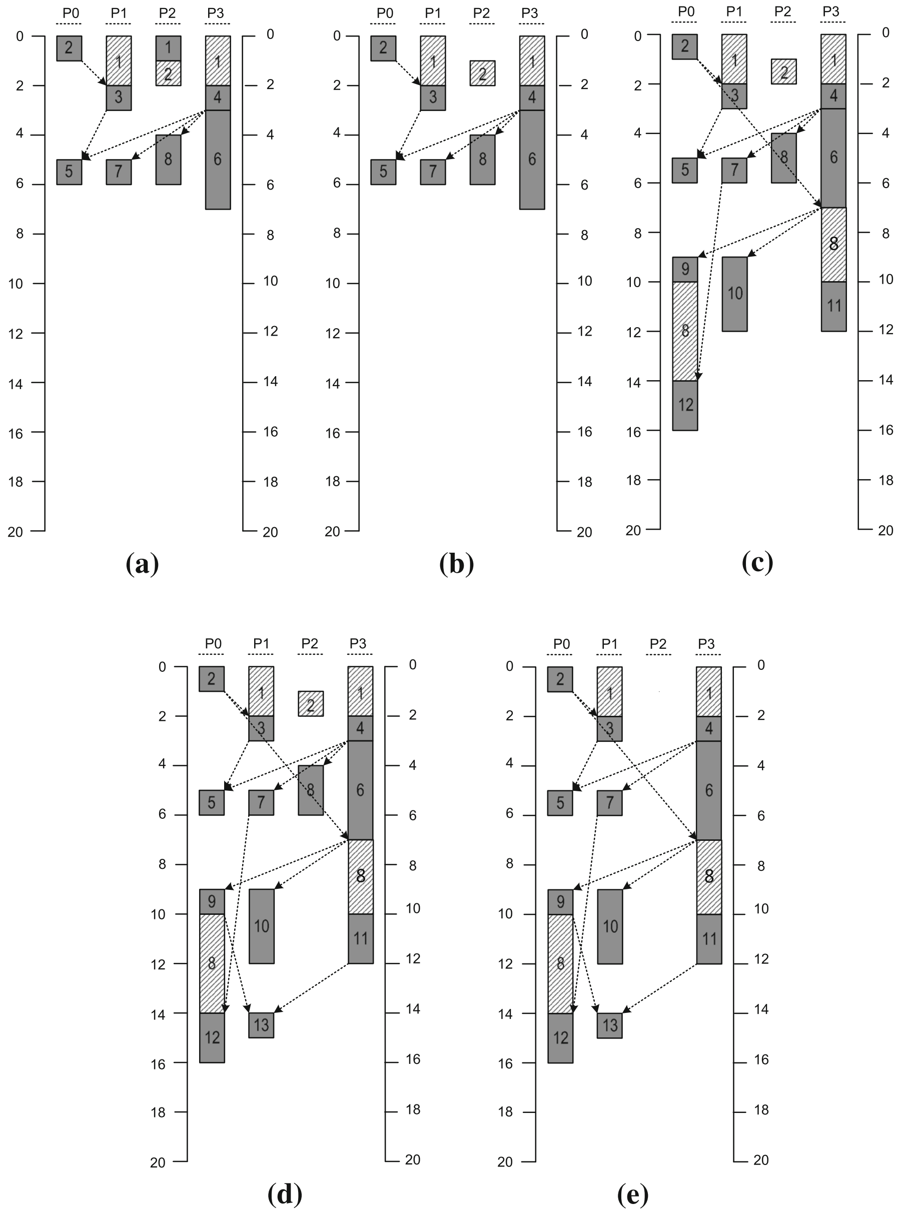
4.3 A scheduling example

To demonstrate the process of RADS, an example schedule of the DAG given in Fig. 1 is demonstrated in Fig. 3. The priorities of tasks are calculated by Eq. (3), and the task queue is constructed using the priorities, which is $\{v_1,v_2,v_3,v_4, v_6,v_8, v_7,v_5,v_9,v_{10},v_{11},v_{12},v_{13}\}$. Tasks are selected one by one from the queue and mapped to their most proper processors. When there is a task in which all children have been scheduled, RADS turns into the redundancy deletion phase to determine if this task has redundant copies. After that, the algorithm returns to the first phase to map tasks again. The process is repeated until all tasks are scheduled. In the example, the children set of task v_1 is denoted by $child(v_1) = \{v_3, v_4, v_5, v_6, v_7, v_8\}$. It is easy to know that the tasks in $child(v_1)$ are all scheduled when the assignment of v_5 is determined. Till now, the generated schedule is shown in Fig. 3(a). According to the rules, the RADS algorithm enters the redundancy deletion phase and the copy (v_1, p_2) is judged to be redundant and is removed from the generated schedule. A new schedule is formed, which is shown in Fig. 3(b).

Next, tasks v_9, v_{10}, v_{11} and v_{12} are mapped one by one. After v_{12}, the child set of v_2 becomes empty, and the algorithm enters the redundancy deletion phase again. At this phase, no redundant copy is found. At last, when all tasks are scheduled, the child sets of all tasks become empty, so all of them are judged in the second phase. The final schedule is given as in Fig. 3(e), and both of (v_8, p_2) and (v_2, p_2) are removed from the original schedule shown in Fig. 3(d). It is apparent that the duplicates generated by RADS is three less than the schedule given in Fig. 2.

4.4 Time complexity of RADS

The time complexity of RADS is expressed in terms of the number of nodes $|V| = n$, the number of edges $|E|$, the number of processors $|P| = m$, and the in/out degree of each task d_{in}/d_{out}, where $\sum d_{in} = \sum d_{out} = |E|$. The complexity of task priority queue generating phase is $O(|E| + |V| \log |V|)$. The complexity of computing the est of each task is $O(d_{in})$, so the complexity of computing the est of all tasks is $O(|E|)$. The complexity of computing the dat of each task on a given processor is $O(d_{out})$, so the complexity of computing the dat of all tasks is $O(|E|)$. The complexity of finding a suitable hole for duplicated task is $O(|V|)$, so the complexity of the duplication operation of all tasks on a processor is $O(|V|^2)$. Because the est of each task must be



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

Fig. 3 A running trace of the RADS algorithm

calculated two times for each processor, one for nonduplication and one for duplication, the complexity of task-mapping and duplicating phase is $O((3|E| + |V|^2)|P|)$.

When all children of a task have been assigned, Algorithm 3 is called to search the redundancy of tasks in Θ. The complexity of Algorithm 3 is $O(\sum_{v_t \in \Theta} \max(d_{out}, |P|)) = O(|\Theta||P|)$. The number of children of a task is equal to its out-degree d_{out}, and the number of duplicated copies of this task is no more than d_{out}. The worst-case complexity of the duplication deletion phase is calculated by $O\ (\sum_{i=1}^{|V|} i \times P) = O(|V|^2|P|)$.

Taking into account that $|E|$ is $O(|V|^2)$, the total algorithm complexity is $O(|V|^2|P|) = O(mn^2)$.

5 Optimizing scheme

The RADS algorithm proposed in Sect. 4 aims at deleting redundant copies dynamically before the whole schedule is generated. It allocates each task to the processor, which can lead to the earliest finish time, and belongs to the class of greedy algorithms. In fact, as we analyzed before, the overall makespan relies on the execution of certain critical tasks and slight delay of other tasks does not affect the overall makespan. Based on this phenomenon, a *further optimizing scheduling scheme* (FOS) in terms of resource consumption is proposed for any given valid schedule in this section.

FOS consists of three phases. In each phase, tasks are shifted earlier or later than their scheduled times of the original schedule, which is to convert task copies to redundant ones. To guarantee the feasibility of a schedule, the following conditions should hold for each task v_i.

- There exists at least one copy of each parent which can provide data for (v_i, p_k).
- The fixed copy of v_i could provide data for all children without local parent v_i.
- The schedule length of each processor should not exceed the overall makespan.

Generally, existing scheduling algorithms adopt the greedy mapping mechanism. It means that tasks are finished as early as possible. To achieve this goal, the $miip$ of each task is to be duplicated, so a great number of duplicated copies are generated. Actually, the finish time of a task is mainly determined by its $miip$, so it is unnecessary that all parents are finished at the earliest time. Figure 2 gives a duplication-based schedule. From the figure we can see that many tasks, such as v_{13}, v_{10}, v_8, can be shifted to a later time, which does not affect the overall makespan. Through shift, the duplicated parents of some tasks become redundant because the communication time allowed is long enough now.

Next, we discuss how to convert a task copy into a redundant one. Before further discussion, some definitions are given first.

Definition 4 The *latest finish time* $lft(v_i, p_k)$ of a task v_i on p_k is the latest time when the copy (v_i, p_k) can be shifted so that dependencies between v_i and its immediate children are preserved.

The first phase of FOS is to calculate lft of all copies and then to shift tasks to their lft. If the lft of task v_i on processor p_k is calculated to be ∞, the copy (v_i, p_k) is determined as a redundancy and is deleted.

In the second phase, the tasks which have multiple copies are shifted to start as early as possible, to break the dependencies between them and their children and hence generate redundancy further. To formalize the start time requirement of a copy, the following definition is given.

Definition 5 The *earliest start time* $est(v_i, p_k)$ of a task v_i on processor p_k is the earliest time that v_i receives data from all of its parents and is ready for execution on p_k.

After the copies of a task are shifted earlier, the time intervals between them and their children get longer. A child which relies on the local parent could receive data from its off-processor parents; hence the local dependency is broken and the local parent could be deleted from a schedule.

In the last phase, we try to migrate tasks between processors. The phase aims at migrating tasks to another suitable processor, and hence their local parents become redundant.

In the following subsections, we described the three phases in detail.

5.1 Phase 1: compute $lfts$, shift tasks, and remove redundancy

The aim of the first phase is to calculate the latest finish time of each task copy, to shift task copies to finish as late as possible, and to decide and delete redundant copies. The pseudo-code of Phase 1 is shown in Algorithm 4.

Algorithm 4 Compute $lfts$, shift tasks, and remove redundancy

```
Require: A schedule $S$ generated by an arbitrary duplication-based algorithm
Ensure: The schedule $S$ after deleting redundant copies
$\sigma$ ←the makespan of the input schedule $S$
for each task $v_i$ in nondecreasing order of $rank$ do
  for each processor $p_k$ that has a copy of task $v_i$ do
    initialize $lft(v_i, p_k) \leftarrow \sigma$
    if $p_k = \vartheta(v_i)$ then
      calculate $lft(v_i, p_k)$ by Eq. 10)
    else
      calculate $lft(v_i, p_k)$ by Eq. 9)
    end if
    if $lft(v_i, p_k) = \sigma$ and $v_i$ is not an exit task then
      delete $(v_i, p_k)$ from $S$
    else
      shift $(v_i, p_k)$ to $lft(v_i, p_k)$
    end if
    update fixed copy of task $v_i$
  end for
end for
```

The input of Phase 1 (see Algorithm 4) is a duplication-based schedule S, which represents the scheduling information of all tasks. Let $(v_i, p_k, st(v_i, p_k), ft(v_i, p_k))$ be an element of schedule S, which means that task v_i is assigned to p_k and its execution starts at time $st(v_i, p_k)$ and finishes at time $ft(v_i, p_k)$. According to Definition 3, it is either a fixed copy or a non-fixed copy. If it is a fixed copy, it must provide data for all the off-processor children of v_i; otherwise, it just needs to offer data for its local children. Due to the difference, the lfts can be calculated in two situations, which are introduced as follows.

Let $V_{lc}(v_i, p_k)$ denote the local child set of v_i on p_k and $V_{oc}(v_i, p_k)$ denote the children of v_i without local duplication of v_i. If $p_k \neq \vartheta(v_i)$, it means that (v_i, p_k) is a non-fixed copy and has local children $V_{lc}(v_i, p_k)$. So, (v_i, p_k) just needs to provide data for its local children, and its latest finish time can be calculated by

$$lft(v_i, p_k) = \min_{(v_c, p_k) \in V_{lc}(v_i, p_k)} \{st(v_c, p_k)\}. \tag{9}$$

If $p_k = \vartheta(v_i)$, it is the fixed copy and must provide data for all its off-processor children in $V_{oc}(v_i, p_k)$ and its own local children in $V_{lc}(v_i, p_k)$. Thus,

$$\begin{aligned} lft(v_i, p_k) = \min\{ & \min_{(v_c, p_k) \in V_{lc}(v_i, p_k)} st(v_c, p_k), \\ & \min_{(v_c, p_r) \in V_{oc}(v_i, p_k)} (st(v_c, p_r) - c_{i,c})\}. \end{aligned} \tag{10}$$

In this phase, the tasks are considered in nondecreasing order of ranks, which makes sure that all children have been shifted before their parent task. Let v_i denote the task being considered in the particular iteration of the for loop shown in lines 4-4 of Algorithm 4, and $\xi(v_i)$ denote the fixed copy of v_i. In the phase, the lfts of all v_i's copies are initialized as σ. σ is the makespan of the input generated. This means that our optimizing scheme will not deteriorate the performance of the original schedule. According to the above analysis, if the lfts of task copies are calculated to be σ and they are no exit tasks, this indicates that those task copies do not need to provide data for any children; therefore, they can be removed from the schedule. To distinguish redundancy from exit tasks, we adopt the method as follows. If v_i is not an exit task and $lft(v_i, p_k) = \sigma$, (v_i, p_k) is determined to be a true redundant copy. At the end of the for loop, we delete redundancy from S and update the fixed copy and the corresponding information.

Consider an example input schedule shown in Fig. 2. There are 13 tasks scheduled on four processors and the arrows in the schedule show important off-processor children dependencies. From the figure, we can see that tasks v_1, v_2, and v_8 have multiple copies, which are the potential redundancy. According to Algorithm 4, tasks are traversed in the nondecreasing order of $rank$. Firstly, those tasks with single copy are shifted to their lfts calculated by Eq. (10) or Eq. (9). Considering v_8, since both of its children v_{12} and v_{11} have local parent on p_0 and p_3, the copy of (v_8, p_2) has neither off-processor children nor local children, so $lft(v_8, p_2)$ is calculated as 16. Because v_8 is not an exit task and $\sigma = 16$, (v_8, p_2) is removed from the schedule. The processing procedures of v_2 and v_1 are similar to that of v_8. (v_2, p_2) and (v_1, p_2) are also deleted from the schedule. The schedule processed by Phase 1 is shown in Fig. 4(a).

5.2 Phase 2: compute *est* and shift tasks backward

Phase 2 is to calculate the earliest start times of the tasks with multiple copies and to shift them to start as early as possible. This aims at lengthening the interval between

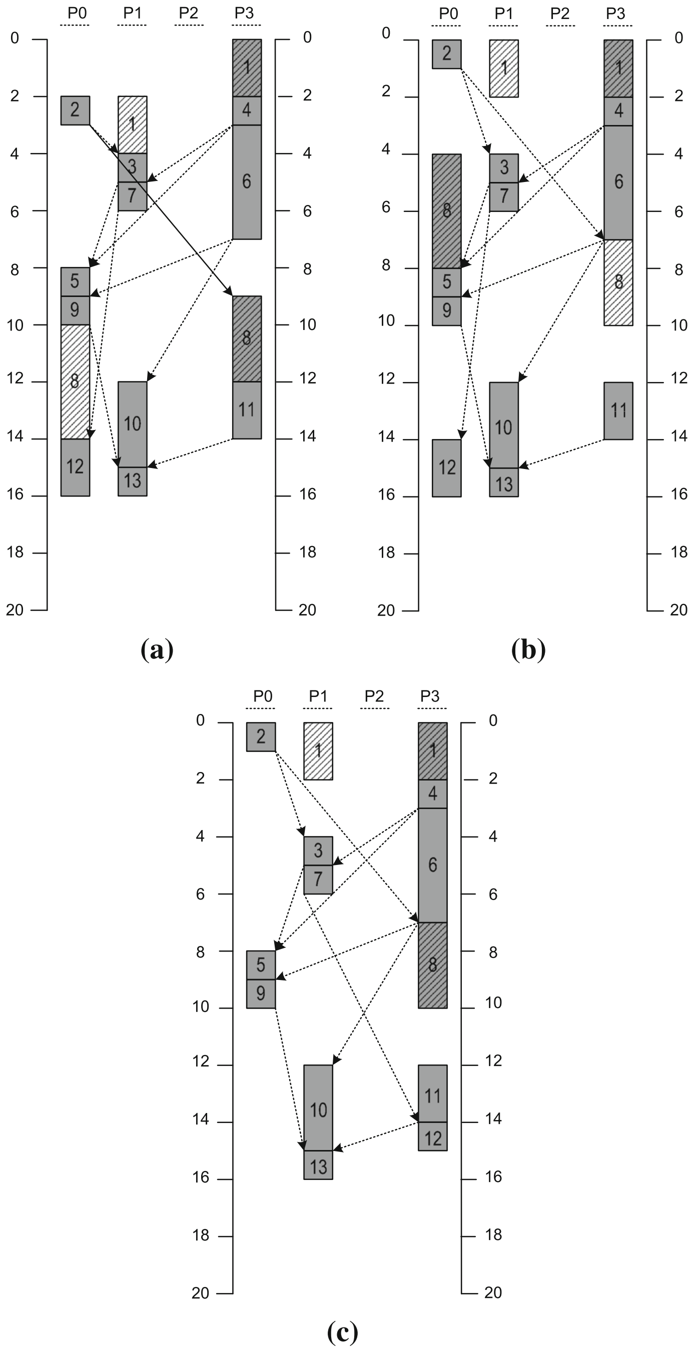


Fig. 4 A running trace of the optimizing scheduling

a task and its children, which makes preparation for the merging operation in Phase 3. The pseudo-code of Phase 2 is described in Algorithm 5.

Algorithm 5 Compute est, shift tasks with multiple copies

```
Require: A schedule $S$ generated by Phase 1
Ensure: The schedule $S$ after shifting tasks
 1: for each task $v_i$ in nonincreasing order of $rank$ do
 2:   if $v_i$ has multiple copies then
 3:     for each parent $v_p$ of $v_i$ do
 4:       for each processor $p_j$ having a copy of $v_p$ do
 5:         $est(v_p, p_j) \leftarrow 0$
 6:         calculate $est(v_p, p_j)$ by Eq. (11)
 7:         shift $(v_p, p_j)$ to $est(v_p, p_j)$
 8:         update the fixed copy of task $v_p$
 9:       end for
10:     end for
11:     for each processor $p_k$ having a copy of $v_i$ do
12:       $est(v_i, p_k) \leftarrow 0$
13:       calculate $est(v_i, p_k)$ by Eq. (11)
14:       shift $(v_i, p_k)$ to $est(v_i, p_k)$
15:       update the fixed copy of task $v_i$
16:     end for
17:   end if
18: end for
```

In the schedule processed by Phase 1, each child gives preference to its local parent to receive data. In fact, if another parent copy on a different processor can provide the needed data instead of the local parent, the child can receive data from the off-processor parent instead of the local parent, and then the local parent can be removed from a schedule. To convert local constraints into off-processor constraints, the best method is to bring forward the execution of the parents as early as possible. To calculate the earliest start time of v_i on p_k, the data arrival times of all its parents must be known. Let v_p be a parent of task v_i. If (v_i, p_k) has a local copy of v_p, it receives data from the local copy (v_p, p_k) without communication; otherwise, it receives data from the fixed copy of v_p. The est of v_i on p_k is calculated by:

$$\begin{aligned} est(v_i, p_k) = \max\{ & \max_{(v_p, p_k) \in V_{lp}(v_i, p_k)} ft(v_p, p_k), \\ & \max_{(v_p, \vartheta(v_p)) \in V_{op}(v_i, p_k)} (ft(v_p, \vartheta(v_p)) + c_{p,i})\}. \end{aligned} \tag{11}$$

where $V_{lp}(v_i, p_k)$ is the local parent set of (v_i, p_k), and $V_{op}(v_i, p_k)$ is the fixed copies of the off-processor parents of (v_i, p_k).

Let v_i be the current task being considered that has multiple copies. Because the finish time of v_i is determined by its parents, its parents must be processed before v_i. In the loop shown in lines 3–10 of Algorithm 5, the est of each parent copy (v_p, p_j) is calculated and (v_p, p_j) is shifted to start as early as possible. After all parents of v_i have been processed, in the loop shown in lines 11–16, $est(v_i, p_k)$ of v_i on each p_k is calculated and (v_i, p_k) shifted backward. It is noticed that a task is allowed to jump over another task on the same processor.

Figure 4(b) shows the schedule in our running example at the end of this phase. v_1 is the first processed task which has two copies. Next, v_8 is considered and it has two parents, v_2 and v_4, respectively. According to our algorithm, both v_2 and v_4 must be shifted before v_8. By calculation, the copy of v_2 on p_0 is shifted to start at 0. After all parents of v_8 are processed, the ests of v_8 on p_0 and p_3 can be calculated, respectively. From the example, we can see that v_8 on p_0 jumps over v_5 and v_9 and starts at time 4.

5.3 Phase 3: merge local children and remove redundancy

In Phase 2, the multi-copy tasks are brought forward, so the distances between them and their children become longer than before. The children which depend on their local parents before can receive data from off-processor parents now; hence, the local parents become redundant and can be deleted. In addition, we consider migrating a local child of a task copy to another processor if it has enough idle time. By doing this, a task copy without local children may be removed. According to the above analysis, it can be found that there are several situations in which redundant copies can be produced. Next, we will discuss them in detail.

Algorithm 6 outlines Phase 3. The tasks are traversed in the nondecreasing order of $rank$. Let v_i be the first task to be considered. To find out all redundancies of task v_i, we group all copies of v_i in pairs, and each possible pair is assigned with a priority based on the execution time difference of the two copies in the pair. Let $< r, s >$ be a pair of copies of task v_i which are assigned to processor p_r and p_s. The execution time difference of $< r, s >$ is calculated by

$$diff^i_{r,s} = w_{i,s} - w_{i,r}. \tag{12}$$

A pair with greater execution time difference is assigned with higher priority. In line 2, all pairs of task v_t are inserted into a queue Q in a nonincreasing priority. The first pair is selected from Q and is denoted by $< p_k, p_l >$, where p_l is called the original processor and p_k is called the objective processor. For each pair $< p_k, p_l >$, we try to decide if (v_t, p_l) can be deleted with the help of (v_t, p_k). The possible situations are discussed in lines 5–27 of Algorithm 6.

The copy of (v_t, p_l) can be deleted if the precedence constraints of all tasks are still satisfied after (v_t, p_l) is deleted. If (v_t, p_l) has no local child (see line 5), it is for sure the fixed copy of v_t and can provide data for all of its off-processor children. In line 7, $est(v_t, p_l)$ is calculated. If v_t is not an entry task yet, $est(v_t, p_l) = 0$, according to Eq. (11), it is concluded that other copies of v_t could provide data for all its children. So, (v_t, p_l) is redundant and can be deleted. If (v_t, p_l) has local children, lines 5–10 give three situations to decide if (v_t, p_l) is redundant. Let v_i be the current local child being considered. If the copy (v_i, p_l) can receive data from another copy of v_t on a different processor, (v_i, p_l) is unnecessary to migrate and the algorithm turns to consider the next child; otherwise, we determine if (v_i, p_l) can be shifted to the objective processor p_k and the steps are shown as follows. The est and lft of v_i on

Algorithm 6 Merge local children of tasks with multi-copy

```
Require: A schedule $S$ generated by Phase 2
Ensure: The schedule $S$ after merging
1:  for each task $v_t$ that has multiple copies in nondecreasing order of $rank$ do
2:    $Q \leftarrow \{< p_l, p_k > | (v_t, p_l), (v_t, p_k) \in Q\}$
3:    for all $< p_l, p_k > \in Q$ do
4:      $flag \leftarrow true$;
5:      if $V_{lc}(v_t, p_l) = \emptyset$ then
6:        $est(v_t, p_l) \leftarrow 0$
7:        calculate $est(v_t, p_l)$ by Eq. (11)
8:        if $est(v_t, p_l) = 0$ and $v_t$ is not an entry-task then
9:          delete $(v_t, p_l)$
10:       end if
11:     else
12:       for all $v_i$ in $V_{lc}(v_t, p_l)$ do
13:         if another copy of $v_t$ can provide data for $v_i$ then
14:           continue;
15:         end if
16:         calculate $est(v_i, p_k)$ and $lft(v_i, p_k)$
17:         if there exists $v_i$ on $p_k$ during interval $[est(v_i, p_k), lft(v_i, p_k)]$ then
18:           delete $(v_i, p_l)$; continue;
19:         else
20:           find available idle slot in $p_k$ for $v_i$ during interval $[est(v_i, p_k), lft(v_i, p_k)]$
21:           if proper idle slot exists then
22:             delete $(v_i, p_l)$ and insert $v_i$ into $p_k$
23:           else
24:             $flag \leftarrow false$; break;
25:           end if
26:         end if
27:       end for
28:     end if
29:     if $flag = true$ then
30:       $Q \leftarrow Q - \{< p_i, p_j > |\text{for all } i = l, \text{ or } k = l\}$
31:     end if
32:   end for
33: end for
```

the objective processor p_k are calculated in line 16. The constraints are satisfied only when v_i is scheduled on p_k during interval $[est(v_i, p_k), lft(v_i, p_k)]$. If there has been a copy of v_i on p_k during the interval, we delete (v_i, p_l) from processor p_l; otherwise, we search a proper idle slot on p_k for v_i to insert. If the proper idle period cannot be found, (v_t, p_l) cannot be deleted and the flag is set as false. After the if-then-else statement in lines 5–27 is finished, if the flag value is equal to $true$, it means all local children of (v_t, p_l) are shifted to processor p_k and the original copy (v_i, p_l) is deleted. Finally, we remove all pairs related to p_l from Q if (v_i, p_l) is deleted. Now, the merging of a pair is complete and the next pair starts.

Fig. 4(c) shows the schedule after Phase 3. v_8 is the first considered task and its pair queue is $Q = \{< p_3, p_0 >, < p_0, p_3 >\}$. Because (v_8, p_0) has only one local child v_{12}, and (v_8, p_3) cannot satisfy the dependency with v_{12}, v_{12} is attempted to be shifted to p_3. The est and lft of v_{12} on p_3 are 14 and 16, respectively. The idle slot [14, 16] on p_3 is available for v_{12}. Hence, the shift is successful and (v_8, p_0) is deleted from the schedule.

5.4 Time complexity of FOS

Complexity of FOS is expressed in terms of the number of nodes $|V|$, the number of edges $|E|$, the number of processors $|P|$, and the in/out degree of each task d_{in}/d_{out}.

In lft computation of Phase 1, all copies of all children of each task are considered, resulting in time complexity of $O(|E||P|)$. Shifting all tasks in a processor requires no more than $O(|V|)$ operations because each task can be executed no more than once on a processor. Since all copies of all tasks are considered, the overall complexity of Phase 1 is $O(|P|(|E||P|+|V|))$=$O(|E||P|^2)$.

To calculate est of one task in Phase 2, the est of its parents must be calculated first. Since all copies of all parents of each task must be considered, and the complexity of calculating each copy is $O(d_{in})$, the complexity for all tasks is less than $O(|E|d_{in}^{max})$, where d_{in}^{max} is the maximum in-degree among all tasks. The shifting operation for all parents requires time $O(|E|)$. Then in calculation of est for all copies of each task, only the fixed copy or local copy of each parent needs to be considered for each copy. So, the complexity of calculating est of all tasks is $O(|V||P|)$. The overall complexity of Phase 2 is $O(|E|d_{in}^{max}+|V||P|)$.

In each round of Phase 3, one pair of a task is considered to merge. The number of elements in Q for each task is $\max(d_{out}^2, |P|^2)$. For each pair (p_i, p_j) of each task v_t, the est and eft of all local children of (v_t, p_i) must be calculated, resulting in time complexity of $O((d_{in}+d_{out}|P|))$. The overall complexity of Phase 3 is $O(|V|(\max(d_{out}^2, |P|^2))(d_{in}+d_{out}|P|)) = O(|V||P|^4)$.

In summary, the overall time complexity of the optimizing scheme is $O(|E||P|^2 + |V||P|^4) = O(n^2m^2+nm^4)$.

6 Experimental results and analysis

We evaluate the performance of the proposed algorithms on random DAGs as well as DAGs from two real applications. The random DAGs are generated with three varying parameters as follows.

- DAG size n: The number of tasks in an application DAG.
- Communication to computation cost ratio CCR: The average communication cost divided by the average computation cost of an application DAG.
- Parallelism factor λ: The number of levels of an application DAG is generated randomly using a uniform distribution with mean value of $\sqrt{n}/\lambda$ and rounded up to the nearest integer. The width is generated using a uniform distribution with mean value of $\lambda\sqrt{n}$ and rounded up to the nearest integer. A low λ leads to a DAG with a low parallelism degree.

In the random DAG experiments, the number of tasks is selected from the set {100, 200, 300, 400, 500}, and both λ and CCR are chosen from the set {0.2, 0.5, 1.0, 2.0, 5.0}. To generate a DAG with a given number of tasks, λ, and CCR, first, the number of levels is determined by the parallelism factor λ, and then the number of tasks at each level is determined. Edges are generated only between the nodes in adjacent levels, obeying a 0–1 distribution. Each task is assigned with a computation cost from a given interval following a uniform distribution. To obtain the desired

CCR for a graph, the communication cost is also randomly selected with a uniform distribution, whose mean depends on the product of CCR in {0.2, 0.5, 1.0, 2.0, 5.0} and the average computation cost.

We also test the algorithms on task graphs from Gaussian elimination (GE) and molecular dynamic code (MDC) applications. For these applications, because the shapes of the DAGs are deterministic, we only investigate the impacts of CCR and the number of used processors on the performance. The values of computation and communication cost are generated using the same method as the random DAG experiments. For each combination of parameter values and for each DAG type, the experiments are repeated 50 times to avoid scattering effects. The results are averaged over all tested values.

Experimental results for RADS and RADS+FOS are presented in comparison to HLD [12]. The performance measures adopted in the experiments are makespan and resource consumption, which are introduced in Sect. 3.3. Since resource consumption of applications varies with the number of tasks and has a large variation range, it is necessary to normalize the resource consumption. Here, we define the *normalized resource consumption* (NRC) as a metric measuring resource consumption:

$$NRC = \frac{Pbt(S)}{Pbt_{\text{lower}}}, \tag{13}$$

where $Pbt(S)$ is the resource consumed by a schedule S, and Pbt_{lower} represents the absolute lower bound on the resource consumed by an application. The calculation of Pbt_{lower} is given by

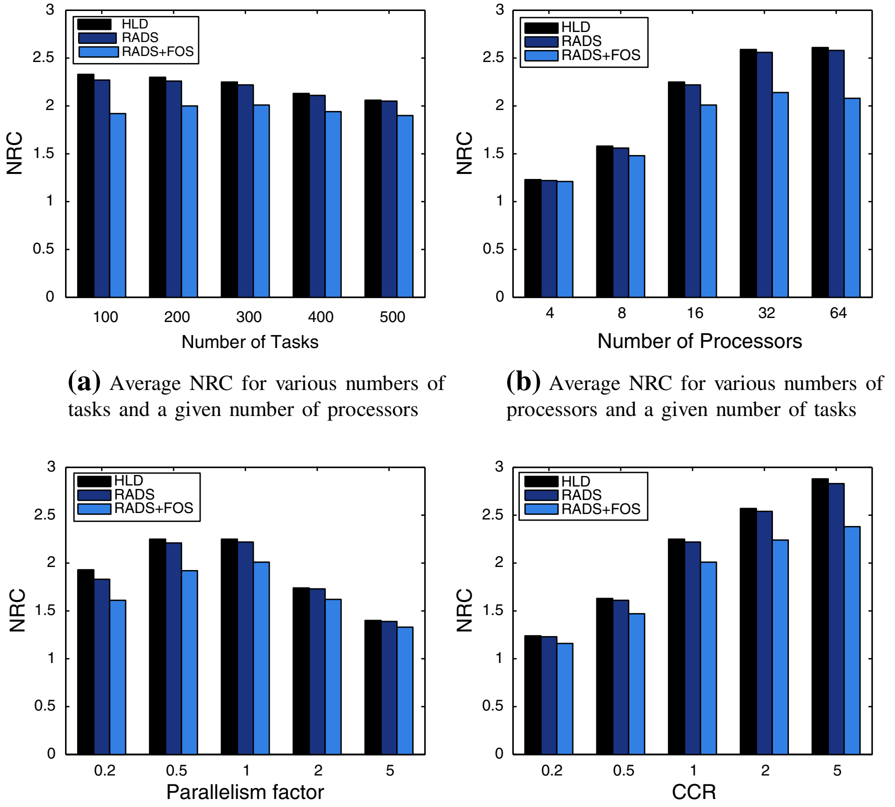
$$Pbt_{\text{lower}} = \sum_{i=1}^{n} \min_{j=0}^{m-1} w_{i,j}. \tag{14}$$

6.1 Randomly Generated DAGs

In this subsection, we conduct performance comparison of the three scheduling algorithms. In Fig. 5, the NRC obtained by each algorithm while varying the number of tasks, the number of processors, parallelism factor, and CCR are presented. Notice that since NRC is simply an upper bound on the resource consumption, an NRC value greater than 1 does not indicate that the schedule has not improved.

The performance of the algorithms in terms of resource consumption is compared with respect to various graph characteristics and different numbers of processors. From Fig. 5, it is known apparently that both RADS and RADS+FOS perform better than the HLD algorithm, and RADS+FOS provides the smallest NRC on average.

The first set of experiments compare resource consumption of the algorithms with respect to various graph sizes (see Fig. 5(a)). The average NRC value of RADS on all generated graphs is reduced by 3 % compared with the HLD algorithm. When combining with FOS, the ratio is up to 12 %. From Fig. 5(a), we can notice that the difference of NRC between RADS+FOS and HLD decreases with the increasing number of tasks. The explanation is as follows. As the number of tasks scheduled on a fixed number of processors increases, tasks become more prone to be assigned



(a) Average NRC for various numbers of tasks and a given number of processors

(b) Average NRC for various numbers of processors and a given number of tasks

(c) Average NRC for various values of λ and a given number of tasks

(d) Average NRC for various values of CCR and a given number of tasks

Fig. 5 Average NRC of random DAGs

to the same processor with their parents, so less tasks are duplicated, which reduces the chance of duplication removal and improvement in resource consumption of the proposed algorithms.

Figure 5(b) shows the experimental results with respect to different numbers of processors. Similarly, both RADS and RADS+FOS outperform the HLD algorithm. We can observe that the average NRCs increase when the number of processors increases from 4 to 32, and is constant from 32 to 64. This is because duplication-based algorithms are prone to duplicating more tasks when there are enough idle processors, which leads to an increasing number of duplications and hence an increasing number of redundant copies. According to the experimental results, RADS reduces resource consumption by 1.93 % on average compared with HLD, while RADS+FOS reduces by 11.25 % on average, and the ratio is up to 20.24% with 64 processors.

In the third set of experiments, the average NRCs produced by the three algorithms are measured with various parallelism factor λ. Fig. 5(c) shows that the average NRCs increase with the increase in λ at the beginning, reach a peak at $\lambda = 1.0$, and then decrease gradually from 1.0 to 5.0. Moreover, the NRC provided by RADS is

Table 4 A comparison of makespan for random DAGs

Parameter	Performance		
	Shorter	Equal	Longer
Number of tasks	86	151	13
Number of processors	93	138	19
Parallelism factor	76	151	23
CCR	126	94	30

improved by 5.22, 2.14, 1.41, 1.03, and 0.61 % compared with the HLD algorithm for $\lambda = 0.2, 0.5, 1.0, 2.0, and 5.0$, respectively. RADS+FOS reduces the resource consumption by 16.32, 14.88, 10.66, 7.38, and 4.60 %, respectively. The data show decreased improvement as λ increases. With a fixed number of tasks and processors, as λ gets smaller, the generated tasks have smaller parallelism. There are enough idle period on the processors to duplicate tasks, which benefit RADS and FOS. As λ increases, the feature becomes weaker, which deteriorates the performance improvement of RADS and FOS.

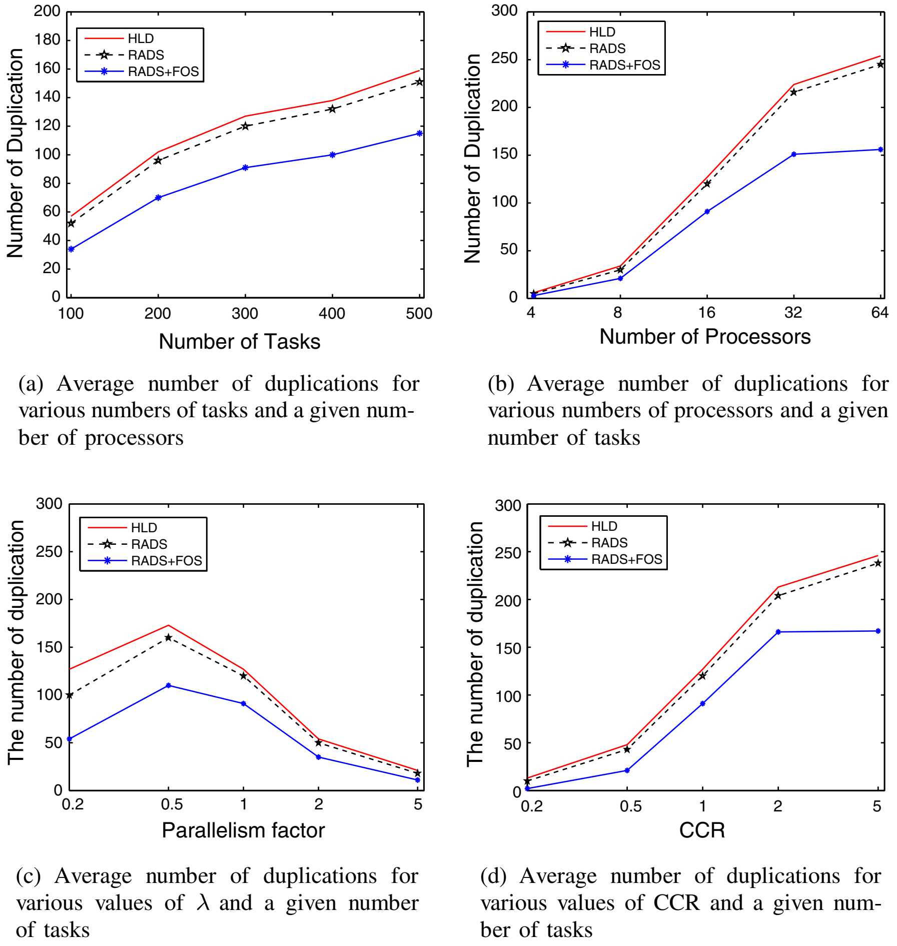
The last set of experiments aim at studying resource consumption of the three algorithms with respect to various CCR values. It is clear from Fig. 5(d) that the average NRCs get greater with increasing CCR. When CCR increases, the ratio of communication and communication cost increases, and the communication cost dominates the computation cost when CCR > 1. Tasks are assigned repeatedly to eliminate the communication between tasks, which is the reason that NRCs increase; hence the performance gaps between RADS and HLD, and RADS+FOS and HLD increase.

To present the performance of our algorithms better, we give four more groups of data shown in Fig. 6, which aim at giving a clear picture on how many duplications are deleted by our algorithms. By using the proposed algorithms, we can obtain a remarkable reduction in the number of duplications for all combinations of parameters. These curves exhibit similar characteristics to those in Fig. 5, because more duplications lead to greater resource consumption.

Since makespan is an important measure to evaluate the performance of algorithms, we count the number of times that a schedule generated by RADS has shorter, equal, and longer makespan compared with that generated by HLD, listed in Table 4. From the table, we can see that the percentage for RADS that outperformed HLD in terms of makespan was 38.1 %, and the percentage for which the schedules generated by the two algorithms had the same makespan was 53.4 %. Overall, our algorithm improves the performance in terms of makespan compared with HLD. Nevertheless, the improvement is very small, as existing duplication-based algorithms have already achieved strong performance in terms of makespan compared with listing scheduling algorithms.

6.2 Application graphs of real-world problems

In addition to randomly generated task graphs, we also consider application graphs of two real-world problems, namely, the Gaussian elimination algorithm [28,29] and a molecular dynamics code given in [30].



(a) Average number of duplications for various numbers of tasks and a given number of processors

(b) Average number of duplications for various numbers of processors and a given number of tasks

(c) Average number of duplications for various values of λ and a given number of tasks

(d) Average number of duplications for various values of CCR and a given number of tasks

Fig. 6 Average number of duplications of random DAGs

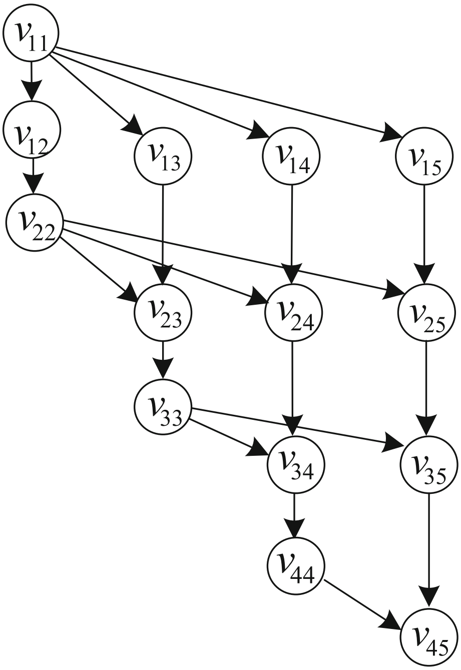
6.2.1 Gaussian elimination

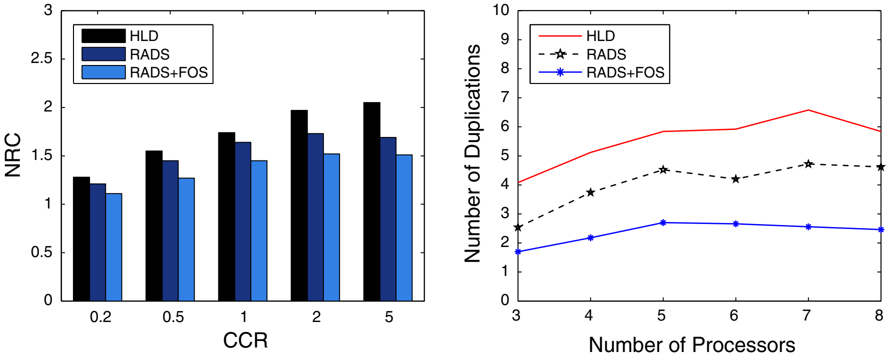
Gaussian elimination is used to determine the solution of a linear system of equations [28]. In this subsection, we consider the schedule of Gaussian elimination solving a 5×5 matrix. The DAG is shown in Fig. 7.

Since the structure of the application graph is known, it is unnecessary to consider those parameters such as the number of tasks and parallelism factor. For the experiments of Gaussian elimination, CCR values and the number of processors are the two factors to be studied. The same CCR values in {0.2, 0.5, 1.0, 2.0, 5.0} are used, and the number of processors is set as three to eight. The experimental results are shown in Fig. 8.

Figure 8(a) gives the average NRC values of the algorithms for various CCRs from 0.2 to 5.0 with five available processors. The performance of RADS and RADS+FOS

Fig. 7 Gaussian elimination for a matrix of size 5





(a) Average NRC for various CCR values

(b) Average number of duplications for various CCR values

Fig. 8 Average NRC and the number of duplications for Gaussian elimination

in terms of resource efficiency is much better than HLD on average. From Fig. 8(a) we can see that RADS has large improvement on HLD which is up to 17.46 %. When combined with FOS, RADS can outperform HLD by up to 26.47 %. Figure 8(b) gives the average number of duplications generated by the three algorithms with various number of processors from 3 to 8. The figure shows that the proposed algorithms generate much less duplications while maintaining the same performance in terms of makespan compared with HLD.

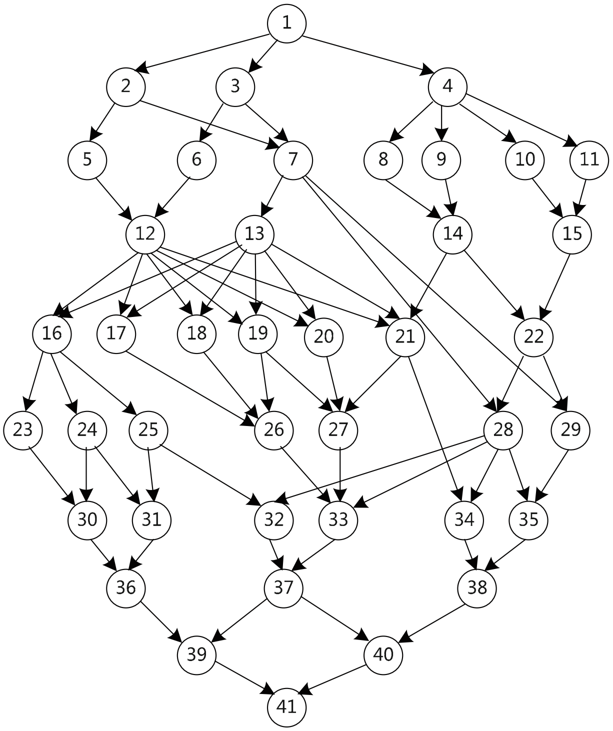
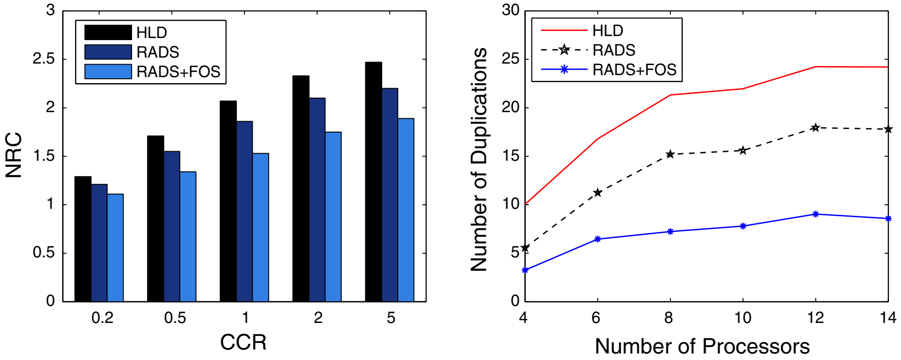


Fig. 9 A molecular dynamics code



(a) Average NRC for various CCR values

(b) Average number of duplications for various CCR values

Fig. 10 Average NRC and number of duplications for molecular dynamic code

6.2.2 Molecular dynamic code

Figure 9 is the task graph of a modified molecular dynamic code given in [30]. Since the number of tasks is fixed and the structure is known, only CCR values and number of processors are considered in our experiments. The number of processors in our experiments is varied from 4 to 14 in steps of 2, and the same CCR values in $\{0.2, 0.5, 1.0, 2.0, 5.0\}$ are used. Fig. 10 shows the experimental results. Fig. 10(a) is with respect to five different CCR values when the number of processors is set as eight. On average, the NRC ranking is HLD, RADS, RADS+FOS. From the figure we can see that both RADS and RADS+FOS algorithms outperform the HLD algorithm. For example, RADS consumes 8.86 % less resource than HLD on average, and RADS+FOs reduces 21.59 % resource consumption on average.

Figure 10(b) presents the number of duplications with respect to six different numbers of processors when CCR is fixed to 1.0. It is concluded from Fig. 10(b) that the duplications generated by RADS are much less than the HLD algorithm. Furthermore, the number of duplications generated by RADS, when combined with FOS, is only $1/3$ of that generated by HLD. Therefore, our proposed algorithms perform very well on resource saving.

In the two groups of experiments for real-world applications, the makespan of our proposed algorithms is much shorter than that of list scheduling algorithms, but there is little improvement compared with duplication-based algorithms. Therefore, we do not give the detailed results of makespan here. In summary, our proposed algorithms are better than the existing duplication-based algorithms.

7 Conclusions

Most duplication-based algorithms duplicate parents for all tasks if the duplication action can lead to an earlier finish time. However, our analysis shows that some duplications are unnecessary. Thus, this kind of duplications, which we call redundant copies, cause a large amount of wasted resource, and even a longer makespan.

In this paper, we propose a resource-aware scheduling algorithm with reduced task duplication on HC systems, which is called RADS algorithm. The algorithm focuses on the elimination of redundant duplications dynamically during the process of scheduling. In the proposed algorithm, when all children of a task have been assigned, the task is reconsidered to determine whether its copies are necessary or not. To improve the performance of RADS, a further optimizing scheme called FOS is proposed. The performance of RADS and RADS+FOS is compared with the HLD algorithm in terms of makespan and resource consumption. The experimental results show that both RADS and FOS perform well on resource efficiency. Although the makespan of our algorithms is not improved noticeably, it is also acceptable because duplication-based algorithms already obtain good performance in terms of makespan.

Future investigation in this area can be performed in the following direction. We will modify the algorithms to take communication contention into consideration. Duplications can reduce the overall makespan on a communication model without contention; however, this does not hold when there is communication contention. Improper duplications would aggravate the contention, which has great negative impact on the performance of algorithms.

Acknowledgments The authors would like to thank the five anonymous reviewers for their constructive comments to improve the presentation of the paper. This research was partially funded by the Key Program of National Natural Science Foundation of China (Grant No. 61133005)and the National Natural Science Foundation of China (Grant Nos. 61070057,61103047,61370095),the Ph.D. Programs Foundation of Ministry of Education of China(20100161110019), the National Science Foundation for Distinguished Young Scholars of Hunan (12JJ1011), the Innovation Fund Designated for Graduate Students of Hunan Province (No. CX2013B142), and the Project of National Natural Science Foundation of China under grant 61202109.

References

1. Freund RF, Siegel HJ (1993) Heterogeneous processing. IEEE Comput 26(6):13–17
2. Maheswaran M, Braun TD, Siegel HJ (1999) Heterogeneous distributed computing. Encycl Electr Electron Eng 8:679–690
3. Cosnard M, Loi M (1995) Automatic task graph generation techniques. In: System Sciences, 1995. Proceedings of the 28th Hawaii international conference on, vol 2. IEEE, pp 113–122
4. Wu MY, Gajski DD (1990) Hypertool: a programming aid for message-passing systems. IEEE Trans Parallel Distrib Syst 1(3):330–343
5. Iverson MA, Ozguner F, Potter LC (1999) Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. In: Heterogeneous computing workshop, 1999 (HCW'99). 8th proceedings. IEEE, pp 99–111
6. Sinnen O (2007) Task scheduling for parallel systems, vol 60. Wiley-Interscience, Hoboken, NY
7. Garey MR, Johnson DS (1990) Computers and intractability: a guide to the theory of NP-completeness. W. H. Freeman & Co., New York, NY
8. Ullman JD (1975) Np-complete scheduling problems. J Comput Syst Sci 10:384–393
9. Radulescu A, van Gemund AJC (2000) Fast and effective task scheduling in heterogeneous systems. In: Proceedings of the 9th heterogeneous computing workshop, 2000. (HCW 2000), pp 229–238
10. Lotfifar F, Shahhoseini HS (May 2009) A low-complexity task scheduling algorithm for heterogeneous computing systems. In: Third Asia international conference on modelling simulation, 2009. (AMS '09), pp 596–601
11. Daoud MI, Kharma N (2008) A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. J Parallel Distrib Comput 68(4):399–409
12. Bansal S, Kumar P, Singh K (2005) Dealing with heterogeneity through limited duplication for scheduling precedence constrained task graphs. JParallel Distrib Comput 65(4):479–491
13. Topcuoglu H, Hariri S, Wu M-Y (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distrib Syst 13(3):260–274
14. Ranaweera S, Agrawal DP (2000) A scalable task duplication based scheduling algorithm for heterogeneous systems. In: Proceedings of the 2000 international conference on parallel processing, pp 383–390
15. Hagras T, Jane brevecek J (2005) A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. Parallel Comput 31(7):653–670
16. Lai K-C, Yang C-T (2008) A dominant predecessor duplication scheduling algorithm for heterogeneous systems. J Supercomput 44:126–145
17. Bansal S, Kumar P, Singh K (2003) An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems. IEEE Trans Parallel Distrib Syst 14(6):533–544
18. Kwok Y-K, Ahmad I (1996) Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. IEEE Trans Parallel Distrib Syst 7(5):506–521
19. Boeres C, Filho JV, Rebello VEF (Oct 2004) A cluster-based strategy for scheduling task on heterogeneous processors. In: 16th symposium on computer architecture and high performance computing, 2004. (SBAC-PAD 2004), pp 214–221
20. Liou JC, Palis (1996) An efficient task clustering heuristic for scheduling dags on multiprocessors. In: Proceedings of parallel and distributed processing symposium
21. Fangfa F, Yuxin B, Xinaan H, Jinxiang W, Minyan Y, Jia Z (2010) An objective-flexible clustering algorithm for task mapping and scheduling on cluster-based noc. In: 2010 10th Russian–Chinese symposium on laser physics and laser technologies (RCSLPLT) and 2010 academic symposium on optoelectronics technology (ASOT), pp 369–373, 28 2010-Aug 1

22. Tang X, Li K, Liao G, Li R (2010) List scheduling with duplication for heterogeneous computing systems. JParallel Distrib Comput 70(4):323–329
23. Zong Z, Manzanares A, Ruan X, Qin X (2011) Ead and pebd: two energy-aware duplication scheduling algorithms for parallel tasks on homogeneous clusters. IEEE Trans Comput 60(3):360–374
24. Shin K, Cha M, Jang M, Jung J, Yoon W, Choi S (2008) Task scheduling algorithm using minimized duplications in homogeneous systems. J Parallel Distrib Comput 68(8):1146–1156
25. Bozdag D, Ozguner F, Catalyurek UV (2009) Compaction of schedules and a two-stage approach for duplication-based dag scheduling. IEEE Trans Parallel Distrib Syst 20(6):857–871
26. Mei J, Li K (2012) Energy-aware scheduling algorithm with duplication on heterogeneous computing systems. In: 2012 ACM/IEEE 13th international conference on grid computing (GRID), IEEE, pp 122–129
27. Khokhar AA, Prasanna VK, Shaaban ME, Wang C-L (1993) Heterogeneous computing: challenges and opportunities. Computer 26(6):18–27
28. Cormen TH, Leiserson CE, Rivest RL (2001) Introduction to algorithms. MIT, Cambridge
29. Cosnard M, Marrakchi M, Robert Y, Trystram D (1988) Parallel gaussian elimination on an mimd computer. Parallel Comput 6(3):275–296
30. Kim SJ, Browne JC (1988) A general approach to mapping of parallel computation upon multiprocessor architectures. In: Proceedings of the international conference on parallel processing, pp 1–8