A Novel Task-Duplication Based Clustering Algorithm for Heterogeneous Computing Environments

Kun He, *Member, IEEE*, Xiaozhu Meng, Zhizhou Pan, Ling Yuan, and Pan Zhou, *Member, IEEE*

Abstract—As a crucial task in heterogeneous distributed systems, DAG-scheduling models a scheduling application with a set of distributed tasks by a Direct Acyclic Graph (DAG). The goal is to assign tasks to different processors so that the whole application can finish as soon as possible. Task Duplication-Based (TDB) scheme is an important technique addressing this problem. The main idea is to duplicate tasks on multiple machines so that the results of the duplicated tasks are available on multiple machines to trade computation time for communication time. Existing TDB algorithms enumerate and test all possible duplication candidates, and only keep the candidates that can improve the overall scheduling. We observe that while a duplication candidate is ineffective at the moment, after other duplications have been applied, this ineffective duplication candidate can become effective, which in turn can cause other ineffective duplications to become effective. We call this phenomenon the chain reaction of task duplication. We propose a novel Task Duplication based Clustering Algorithm (TDCA) to improve the schedule performance by utilizing duplication task more thoroughly. TDCA improves parameter calculation, task duplication, and task merging. The analysis and experiments are based on randomly generated graphs with various characteristics, including DAG depth and width, communication-computing cost ration, and variant computation power of processors. Our results demonstrate that the TDCA algorithm is very competitive. It improves the schedule makespan of task duplication-based algorithms for heterogeneous systems for various communication-computing cost ratios.

Index Terms—Task duplication, clustering and merging, optimal scheduling, heterogeneous environment

1 Introduction

Parallel and distributed computing are prevalent as the scales and complexities of applications are growing rapidly. Applications such as numerical weather prediction, and image processing consume various computing resources, including CPU/GPU, network bandwidth, and memory. A distributed computing system needs a scheduler to effectively allocate these resources to different applications, such that applications can complete as early as possible and computing resources are fully utilized.

Scheduling in a distributed system has been a major research topic due to its importance and difficulty. Different applications may have diverse requirements on computing resources (CPU-intensive or IO-intensive) and various goals (fairness[1], low latency, or high throughput, or real time [2]), making a scheduler difficult to balance all these factors. In general, optimal task scheduling is NP-hard. Thus, various heuristics are proposed to address the need of particular application scenarios. For example, the tasks may arrive at different time periods, representing a dynamic scheduling problem [3]; the tasks may also have requirements on memory, network bandwidth or disk space, representing a bounded-resource scheduling problem [4]; the scheduler may also need to provide fairness to all applications or improve resource utilization, representing a multi-objective optimization problem [5]. This paper focuses on the Directed Acyclic Graph Scheduling (DAG-scheduling), which serves as a foundation for designing scheduling algorithms in more complicated scenarios [6], [7], [8].

DAG scheduling can be classified into the static task scheduling problem [9], [10], [11]. That is, the structure of the application in terms of its task execution times, task dependencies and communication cost is known a priori, and the scheduling can be accomplished statically at compile-time. The DAG model abstracts a distributed application as multiple tasks, where a node in the graph represents a task and an edge represents the execution dependency between two tasks. There are a certain number of processors available to schedule these tasks. We also know the execution time of each task on each processor and the communication time between each pair of dependent tasks if they are allocated to different processors. The goal is to assign all tasks to processors in order to minimize the completion time of the whole application.

The problem of DAG scheduling has been extensively studied [12]. Early works focus on homogenous computing

- *K. He, X. Meng, Z. Pan, and L. Yuan are with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {brooklet60, cherryyuanling}@hust.edu.cn, mxz297@gmail.com, panzhizhou@126.com.*
- *P. Zhou is with School of Electronic Information and Communications, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: panzhou@hust.edu.cn.*

Manuscript received 3 Aug. 2017; revised 27 May 2018; accepted 24 June 2018. Date of publication 28 June 2018; date of current version 12 Dec. 2018. (Corresponding authors: Zhizhou Pan and Ling Yuan.)
Recommended for acceptance by C. Morin.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2851221

environment such as distributed memory machines [13], [14], [15], [16], and recent works pay more attention on heterogeneous systems [17], [18], [19], [20]. Emerging architecture, GPU clusters, are also considered for dense linear algebra computation [21], [22]. Other problem variants include multiple DAGs scheduling [23], hierarchical DAG scheduling [21], stochastic tasks scheduling [24], etc.

For the DAG scheduling techniques, existing algorithms can be classified into three categories, list scheduling [9], [25], [26], [27], cluster-based scheduling [13], [28], [29] and task duplication-based scheduling [30], [31]. And some researchers have combined task duplication with either list scheduling or clustering scheduling to improve the performance [12], [32]. We find that there are two limitations for typical DAG scheduling algorithms. These algorithms rely on key parameters, for example the earliest starting time est. But the calculation of these parameters may be not accurate, leading to a scheduling with suboptimal quality. Second, in existing algorithms, once a task duplication candidate has been evaluated to be ineffective, it will no longer be considered. In our experiments shown in Section 4, however, we observe that an ineffective duplication may become effective again after other duplications have been performed.

In this work, we propose a novel algorithm, called the Task Duplication based Clustering Algorithm (TDCA), to effectively perform task duplication and improve the scheduling quality. TDCA first calculates a set of key parameters with our improved definitions and generates initial task clusters. A task duplication strategy is adopted to modify the initial clusters and new clusters are added when necessary. Then, TDCA merges the clusters to shorten the makespan and reduce the number of occupied processors. In the end, we adopt a task insertion scheme introduced in DCPD [33], which inserts a task to an idle time slot located before its successor task if this insertion could make the successor start earlier. This insertion scheme allows TDCA to generate the optimal solution of makespan 8.0 on the classic EZ benchmark, while the optimal makespan of EZ is regarded as 8.5 in the literature [11]. Note that by default these claims are for plenty of homogeneous resources, and three machines are sufficient for the EZ benchmark.

TDCA contains three main improvements over existing algorithms.

- *New definitions for key parameters*. We redefine several key parameters. For example as compared with our baseline algorithm TANH [34], when defining est, we ignore the communication cost if the task and its predecessor are on the same processor, which yields a more accurate est.
- *Improving the initial clustering*. When producing initial task clusters, existing algorithms typically duplicate the predecessors of a task. On the other hand, TDCA will also consider waiting for transferring the results of a predecessor task from other clusters, which may improve the quality of the initial clusters.
- *Consideration of the chain reaction*. In the task duplication phase, existing algorithms try a list of duplication candidates, applying the candidates that improve the overall scheduling and abandoning the candidates that does not improve the overall scheduling. We identify the chain reaction phenomenon where an ineffective duplication may become effective after other duplications have been applied.

We compare TDCA with three state-of-the-art algorithms, TANH [34], HEFT [19] and DCPD [33]. A random DAG generator is designed to cover different types of DAGs to evaluate how the proposed algorithm fares. We thoroughly investigated the impact of different DAG properties, including number of nodes, communication-computing cost ratio (CCR), number of layers, and heterogeneity of the processors. Experiments on small instances as well as large instances with up to 3000 tasks demonstrate that TDCA can provide more optimization opportunities.

The paper is organized as follows. In Section 2, we discuss the background of this work. The detail of the proposed TDCA is presented in Section 3. Experimental results and analysis are illustrated in Section 4, and conclusion remarks in Section 5.

2 Background

In this section, we first present the formal definition of the DAG scheduling problem, then briefly summarize existing DAG scheduling algorithms, with the focus on the seminal work of TANH [34], which is the base of our proposed Task-Duplication based Clustering Algorithm (TDCA).

2.1 Problem Definition

Assume there are a certain number of heterogeneous processors available to parallel perform computations and communications. There are a set of tasks required to be assigned to the available processors, with each task being an indivisible unit of non-preemptive work. Given the precedence constraints among the tasks, the computation cost of a task on each processor and the communication cost between any two tasks subjected to precedence constraints, the problem is to schedule the tasks on processors to minimize the makespan, defined as the latest completion time of all tasks minus the earliest starting time of all tasks. A feasible schedule should satisfy the following constraints:

1) *Constraints on tasks*: All tasks should be assigned and executed on at least one processor. All the tasks are atomic, meaning that none of them can be interrupted during the execution.
2) *Constraints on communication*: For any pair of tasks subjected to the precedence constraint, the successor cannot start until the predecessor has finished and the resulting data of the predecessor has been transferred to the successor. The transferring time is the communicating cost annotated on the edge between the two tasks in the DAG if these two tasks are not assigned to the same processor, and otherwise 0.
3) *Constraints on processors*: Processors can only run tasks serially, indicating that the execution of any two tasks on the same processor cannot overlap.

We use a DAG $G = \langle V, E, P, T, C \rangle$ to formalize the problem. 1) $V = \{1, 2, \ldots, n\}$ is the set of nodes that represents the tasks. $n \in N$ is the number of tasks. 2) $E = \{\langle i, j \rangle : i, j \in V\}$ is the set of edges, representing the precedence constraints. $e = |E| \in N$ indicates the number of edges. 3)

$P = \{1, 2, \ldots, m\}$ represents the set of available processors, and $m \in N$ indicates the number of processors. 4) T, a two-dimensional $n \times m$ matrix, stands for the computation costs of the nodes. For $i \in V$ and $p \in P$, $T(i, p)$ is the computation time of task i if it is assigned to processor p. 5) C, a two-dimensional $n \times n$ matrix, represents the communication costs. For $i \in V, j \in V$ and $\langle i, j \rangle \in E$, $C(i, j)$ is the time needed to transfer data from i to j when i and j are assigned to different processors.

For convenience, we denote $PRED(i) = \{j : \langle j, i \rangle \in E\}$ as all predecessor tasks of i and denote $SUCC(i) = \{j : \langle j, i \rangle \in E\}$ as all successive tasks of i. If there is more than one entry-node, we add one pseudo entry-node connecting to these entry-nodes. One pseudo exit-node is added in a similar way. Note that the pseudo entry-node and the pseudo exit-node have zero running time on any processor, and the weights of edges connecting the pseudo nodes and the actual start or end nodes are zero.

In practice, partition algorithms [11], [18] are used to transform real-world applications to DAG representations. A DAG is usually assumed to be fully connected [11], [18]. We could also assign a sufficiently large communication cost to certain entries in C if the network is not fully connected [2], [3], [25]. Note that our definitions are suitable for heterogeneous systems, where processors may have different computing power, so that a task may have varied running time on different processors.

2.2 DAG Scheduling Algorithms

We discuss in detail three main categories of the DAG-scheduling algorithms: list scheduling [35], cluster-based scheduling [12], and task duplication-based scheduling [30]. We also discuss a few hybrid algorithms that combine list scheduling with task duplication or combine cluster-based scheduling with task duplication.

List scheduling algorithms generate a task list sorted by the priority of the tasks, which are then sequentially assigned to processors. Variations of such algorithms differ in the method of how to define priorities or how to assign tasks to processors [35]. The main idea of list scheduling is to calculate a priority for each task and then assign tasks with higher priorities first to processors. K. Shin et al. [32] defined three types of task priority, namely S-Level, T-Level and B-Level. S-Level, also called static level, is the length of the longest path from the current node to the exit-node without considering the communication cost among tasks. B-Level, also known as bottom level or downward rank, computes the longest path from the current node to the exit-node considering communication cost. T-Level, namely top level or upward rank, is the length of the longest path from the entry-node to the current node including the communication cost. A weighted average of these three types of priority is used to get a better priority definition [1]. Another type of ranking techniques, called critical path based ranking, assign tasks on a critical path higher priorities. The intuition is that if these critical tasks were postponed, it might delay the entire application. HEFT [19] and PEFT [20] use critical path based ranking methods. List scheduling algorithms are simple to implement, but their solutions are often not as good as the other two types of scheduling algorithms.

Clustering-based algorithms first group the tasks into task clusters and then assign clusters to distinct processors in order to reduce the communication cost [12]. After the initial cluster assignment, clusters may be further merged to reduce the communication cost. The essence of this method is to cluster highly related tasks onto the same processor that communication cost among themselves becomes negligible. If the available number of processors is larger than the number of clusters, their solutions can be very efficient.

Task duplication based algorithms duplicate a task when necessary and assign them to different processors. Therefore, the results of duplicated tasks are readily available on multiple processors, which may help reduce the communication cost. See details in a survey paper [30]. The main idea behind this method is to utilize processor idling time to duplicate predecessor tasks. This may avoid transfer of results from a predecessor through a communication channel.

Hybrid algorithms combine task duplication with either list scheduling or clustering scheduling [9], [18], [27], [33]. To perform task duplication in list scheduling, existing algorithms, such as LDCP [9], HEFD [27], and DCPD [33], often duplicate the predecessor tasks to reduce communication costs. For cluster-based scheduling, such as TDS [18] and TANH [34], task duplication can be performed by allowing an individual task to be assigned to several clusters, which could reduce the communication costs among clusters. Our proposed algorithm belongs to the latter type.

Regardless of the algorithm categories, one common issue is to determine in which order to execute the tasks that have been assigned to the same processors. Two commonly used types of methods are the insertion based method, such as the one used in HEFT [19], and non-insertion based method. Insertion based methods often produce better scheduling because they can make use of potential idle time slots on processors. Suppose that two tasks are scheduled serially on a processor. The latter task may not be able to start running right after the completion of the prior one, because the latter task may have to wait for the completion of other predecessor tasks to get relevant output data. The gap between the two tasks is called an idle time slot. Insertion based methods may insert other tasks to these idle slots to take full advantage of the processors, and thus are more effective. On the other hand, non-insertion based methods are simpler to implement as there is no need to track these idle slots. Our proposed algorithm would adopt an insertion based method to improve scheduling efficiency. HEFT [19] first calculates the task priority according to the B-level, sorts the tasks in non-descending order, and assigns tasks to a processor such that it has the earliest starting time for an initial allocation. HEFT also considers the task insertion scheme.

DCPD [33] dynamically calculates the task priority basing on B-level, T-level and the task computation cost. It gives higher priority to tasks with larger computation cost, longer path to the exit node, and shorter path to the entry-node. The set of unassigned tasks are classified into ready set, partially ready set and unready set. At each iteration, it assigns tasks in the ready set to a processor such that they have the earliest end time. Then, it duplicates the critical predecessor task if the copy makes the predecessor start

earlier. Also, it recalculates the priorities of the unassigned tasks and update the three sets.

2.3 Analysis of TANH

We give a brief description of the TANH algorithm [34], on which we base our proposed algorithm TDCA. TANH has three main steps:

1) *Parameter calculation:* TANH computes the parameters in two stages. First, it calculates the following parameters for each node i in the topological order from the entry node to the exit node: the earliest starting time $est(i)$, the earliest completion time $ect(i)$, favorite processors $fproc(i,1)$ to $fproc(i,m)$, favorite predecessor $fpred(i)$ and B-level $level(i)$. Second, it calculates the following parameters for each node i in the reverse topological order from the exit node to the entry node: the latest allowable starting time $last(i)$ and the latest allowable completion time $lact(i)$.
2) *Initial cluster generation:* Tasks are sorted by the $level$ parameter in the non-decreasing order. At each time, TANH chooses the first unassigned task and add it to a new cluster. Suppose TANH is generating a cluster for task i. TANH iteratively adds one of predecessor task of i into the cluster (in most case, the added task is i's favorite predecessor $fpred(i)$), until the entry node is added into the cluster. The whole cluster is assigned to the first unoccupied processor p, searching in the order from $fproc(i,1)$ to $fproc(i,m)$. If all processors are occupied, this cluster is assigned to a pseudo processor where the runtime of a task is taken as the average runtime of the task.
3) *Task duplication or cluster merging:* If clusters are assigned to pseudo processors in the previous step, TANH performs cluster merging in this step. It merges a cluster with a large task computation cost to a cluster with a small task computation cost, hoping that the merged cluster will only have a small increase in computation cost. The merging process continues until all pseudo processors are merged. On the other hand, if no clusters are assigned to pseudo processors in the previous step, TANH tries task duplication. Specifically, suppose task set $\{x_1, x_2, \ldots, x_k\}$ has been assigned to processor p and $\{x_1, x_2, \ldots, x_k\}$ are in topological order in the corresponding DAG. For $i \leq k$, TANH may tentatively move task x_1 to x_{i-1} to another processor q, which is the first unoccupied processor in the order from $fproc(x_{i-1},1)$ to $fproc(x_{i-1},m)$, and add $fpred(x_i)$, $fpred(fpred(x_i)), \ldots$ up to the entry-node to p. Thereafter, TANH computes the makespan of the new schedule, and it would nullify the above change in case the makespan becomes larger.

While TANH has been shown to be effective in its experiments, we have identified three limitations of TANH that can be improved.

First, definitions of several parameters are rather loose. For example, in our experiments, we have encountered situations where *lact* are negative for some DAGs. These loose parameter definitions may lead to inappropriate decisions on task duplications and sub-optimal schedule.

Second, in the initial cluster generation step, TANH always keeps adding nodes to a cluster until the entry-node is added. However, it is possible that for some task i, there is no need to add any predecessor of i to the cluster, since it could use the computation results from other processors. Our experiments showed that always adding nodes to one cluster until the entry-node may lead to unnecessary duplication, especially when the computation costs are larger than the communication costs.

Third, in the task duplication step of TANH, it does not show the exact order of the positions of performing task duplication. In experiments, we found that duplications conducted in different order may lead to various schedules and the schedules may have significantly different makespans.

3 Proposed Algorithm

We describe the details of the Task Duplication based Clustering Algorithm (TDCA) in this section. The proposed algorithm includes several key parameters and four phases. In the first phase, TDCA assigns tasks to processors to construct the initial clusters. A task duplication method is used to modify the initial clusters in order to shorten the makespan in the second phase. In the third phase, TDCA merges some clusters to obtain a preliminary schedule. And an insertion phase is added to utilize the idle time slot so as to further reduce the makespan. One key difference between TDCA and TANH is that TDCA uses both task duplication and cluster merging to reduce the makespan, while TANH only applies one of the two depending on the relationship between the number of initial clusters and the number of available processors. We also use a DAG example to illustrate the running trace of TDCA. We conclude this section with a complexity and scalability analysis of TDCA.

3.1 Definitions

We redefine many parameters used in TANH [34] to better approximate their values, such as the earliest starting time est and the earliest completion time ect. In addition, we define the critical predecessor trail of a task to improve the task duplication phase.

Definition 1. *Communication Cost. For $j \in PRED(i)$, if j and i are assigned to processor q and p respectively, then the communication cost between the two tasks:*

$$\delta(j,i,q,p) = \begin{cases} 0, & q = p \\ C(j,i), & q \neq p \end{cases} \quad \langle j,i \rangle \in E. \tag{1}$$

Definition 2. *Earliest Starting Time. For $i \in V$ and $p \in P$, $est(i,p)$ indicates the earliest starting time of task i if i is assigned to processor p. $est(i,p)$ can be formulated as follows: $est(i,p) = 0$ for the entry node, and otherwise:*

$$est(i,p) = \max_{j \in PRED(i)} \left\{ \min_{q \in P} \left\{ ect(j,q) + \delta(j,i,q,p) \right\} \right\}. \tag{2}$$

Here we use a $\max$ and $\min$ combination except the entry-node. If task i runs on processor p, the outer $\max$ indicates that all its predecessor tasks must have finished the computation somewhere and have sent the resulting data that i needs to processor p. From the outer $\max$, we could

tell which predecessor blocks task i. On the other hand, since each predecessor of task i could be assigned to any processor in P, the inner min tells which processor is the best choice for predecessor j such that its resulting data could reach processor p as early as possible.

This definition yields a naive method to compute est with a complexity of $O(|E||P|^2) = O(em^2)$. But we can actually reduce the complexity to $O(|E||P|) = O(em^2)$. The key observation is that we do not have to enumerate every processor q to determine the minimum. Since for any j, $ect(j, fproc(j, 1))$ is always the minimum over all $ect(j, q)$. So we only need to find the smaller one between $ect(j, p)$ and $ect(j, fproc(j, 1)) + C(j, i)$.

Therefore, we could simplify the definition and reduce the complexity of computing est from $O(|E||P|^2)$ to $O(|E||P|)$: $est(i, p) = 0$ for the entry node, and otherwise:

$$est(i,p) = \max_{j \in PRED(i)} \Big\{ \min\{ect(j, fproc(j,1)) + C(j,i), ect(j,p)\} \Big\}. \tag{3}$$

Definition 3. *Earliest Completion Time. For $i \in V$ and $p \in P$, $ect(i, p)$ is the earliest completion time of task i on processor p. We define $ect(i, p)$ to be $est(i, p)$ plus the computation cost of task i on processor p:*

$$ect(i,p) = est(i,p) + T(i,p). \tag{4}$$

Definition 4. *The rth Favorite Processor. For $i \in V$ and integer $r \in [1, m]$, $fproc(i, r)$ is the rth favorite processor of task i, which satisfies the following inequality:*

$$ect(i, fproc(i,1)) \leq \cdots \leq ect(i, fproc(i,r)). \tag{5}$$

Note that this definition implies that we can calculate $fproc(i)$ by first calculating all ect and then sorting them in non-decreasing order.

Definition 5. *Critical Predecessor. For $i \in V$, $cpred(i)$ is the critical predecessor of task i and represents the bottleneck predecessor that prevents $est(i)$ from being smaller. Specifically, assuming all tasks are assigned to the favorite processors, $cpred(i)$ is the last predecessor that sends its results to task i.*

$$cpred(i) = \arg\max_{j \in PRED(i)} \Big\{ ect(j, fproc(j,1)) + \delta(j, i, fproc(j,1), fproc(i,1)) \Big\}. \tag{6}$$

Our definition for $cpred(i)$ makes improvement on $fpred(i)$ of TANH in the way that if $fproc(j, 1)$ is the same as $fproc(i, 1)$, we actually do not involve the communication cost between the two tasks.

Definition 6. *Task Priority. We use the B-Level [32] as the task priority. The task priority $level(i)$ for $i \in V$ is the length of the longest path from i to the exit-node in a graph where the weight of node i is the maximum computation cost of i among different processors, and the weight on the edge is the communication cost:*

$$level(i) = \max_{k \in SUCC(i)} \{level(k) + C(i,k)\} + \max_{q \in P}(T(i,q)). \tag{7}$$

Definition 7. *Critical Predecessor Trail. The critical predecessor trail of a task i is defined as $cpred(i)$, $cpred(cpred(i))$, ..., up to the entry-node.*

Intuitively, the critical predecessor trail represents the tasks that may prevent task i from starting earlier. It is used in the task duplication phase to determine which task to duplicate. est, ect, $fproc$ and $cpred$ are calculated in the topological order of the DAG, and $level$ is calculated in the reverse order. Note that we give different definitions on est and $cpred$. Though ect and $fproc$ share the same definitions of TANH, different est leads to distinct ect and $fproc$. Note that we do not use $last$ and $lact$ in TDCA. Only level is exactly the same in both TDCA and TANH.

3.2 Initial Task Clustering

After the calculation on the above parameters, tasks are sorted by their levels in non-decreasing order. TDCA starts to construct the initial clusters. Iteratively, TDCA chooses several tasks and assigns them to the best unoccupied processor. Our task clustering step makes three key improvements over the task clustering step of TANH.

1) If there are more than one predecessors for node i, and $j = fpred(i)$ has not been assigned, TANH just assigns j to the processor p of node i, while TDCA further checks whether j satisfy the following inequality:
$$ect(j,p) \leq ect(j, fproc(j,1)) + C(j,i). \tag{8}$$
If the above inequality fails, we can predict that allocating j on p is less promising than assigning j to its favorite processor and just waiting for the message passing from j to i.
2) If $j = cpred(i)$ is not selected to be in the same cluster of i, i.e., when inequality (8) fails for $cpred(i)$, instead of choosing a predecessor that has the lowest running time on p, we choose an unassigned predecessor k that inequality (9) holds.
$$ect(k,p) \leq ect(k, fproc(k,1)) + C(k,i). \tag{9}$$
If there are multiple predecessors satisfying inequality (9), we then choose the one minimizing $ect(k, p)$. We make the change in order to help i start as early as possible.
3) TANH ends the iteration of task clustering when it reaches the entry-node, while TDCA uses another condition to stop the task clustering. If $j = cpred(i)$ fails to be selected to the same processor, and TDCA cannot find any unassigned predecessor that inequality (9) holds, then it terminates the current clustering.

The details of task clustering are presented in Algorithm 1. We make two comments on this algorithm. First, the inequality introduced above cannot predict accurately whether j is beneficial for i, since ect itself is only an estimation and $fproc(j, 1)$ may be occupied by other tasks. However, experiments in Section 4 demonstrate that it is a good estimation. Second, when the number of tasks n is much larger than the number of processors m, it is possible that there will be no unoccupied processor for unassigned tasks. In such case, we stop the task clustering producer and assign each unassigned task to a processor that minimizes the starting time of the unassigned task, based on the current assignment.

3.3 Task Duplication

In the task duplication phase, we enumerate processors from 1 to m and check every candidate position described as follows. Let $X_p = (x_1, x_2, \ldots, x_k)$ be the tasks $x_1, x_2, \ldots, x_k$ assigned on processor p and they are in topological order in the corresponding DAG. We find a candidate position in the following order.

1) We first check whether there exists an i such that $cpred(x_i) \neq x_{i-1}$, where $i \in [2, k]$. If so, task duplication will be conducted between x_i and x_{i-1}.
2) Then we check whether $x_1 = entry - node$. If not, further task duplication is conducted for the predecessor trail of x_1.

Algorithm 1. Task-clustering Procedure

```
1: Put all tasks into the task array in the non-decreasing order
   of their levels;
2: for every i in the task array do
3:   If task i has been assigned continue
4:   curProc = first unoccupied processor in
5:                fproc(i,1), ..., fproc(i,m);
6:   Assign task i to processor curProc;
7:   Status of task i = assigned;
8:   while i ≠ entry − node do
9:     j = cpred(i);
10:    if in − degree(i) > 1 and (j has been assigned
11:          or ect(j, curProc) > ect(j, fproc(j,1)) + C(j,i)) then
12:      Find k, an unassigned predecessor of i that inequal-
         ity (9) holds
13:            and minimizes ect(k, curProc) and set j = k;
14:      if k not exists break
15:    end if
16:    Add task j to processor curProc;
17:    Status of task j = assigned;
18:    i = j;
19:  end while
20:  Set processor curProc occupied;
21: end for
```

In the first case, TDCA is similar to that of TANH. x_1 to x_{i-1} will be moved to another processor q, which is the first unoccupied processor in the order from $proc(x_{i-1}, 1)$ to $fproc(x_{i-1}, m)$ and the predecessor trail of x_i will be added to p. In the second case, we do not need to move any tasks. Instead, we directly add the predecessor trail of x_1 to p. Then, we compute the makespan of the new schedule and accept if the makespan is shortened. Thereafter, we will try to find the next position to conduct the duplication.

We repeat the above process for K times. The reason for the repetition is explained below and the effectiveness will be justified in Section 4. Empirically, we found that $K = 4$ works well in our experiments. The details of the task duplication are shown in Algorithm 2.

Note that the order we apply task duplication to candidate positions may affect the schedule, where different orders might result in different makespans. Most studies do not directly address this issue. For example, Baskiyar and Dickinson [4] proposed a strategy to duplicate bottleneck tasks that would delay the execution of exit-node, but they did not specify the order if there exist multiple bottleneck tasks.

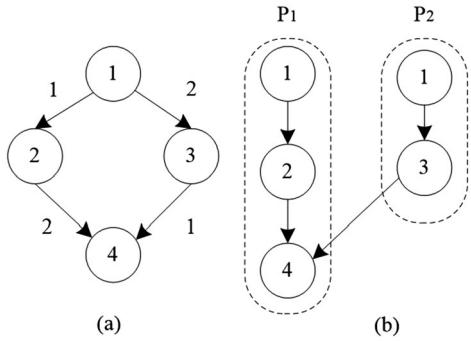


Fig. 1. A DAG example and the TANH schedule.

With experiments, we observe an important fact that previous invalid duplication can become effective again after other duplication has been applied, hence a replication chain would be involved in the whole task duplication procedure. In Section 4, we will give a comprehensive analysis on the phenomenon of replication chains and justify that four times of iterations ($K = 4$) are appropriate to benefit most.

Algorithm 2. Task-Duplication Procedure

```
1: for iterNum from 1 to K do
2:   for processor p from 1 to m do
3:     for i from |X_p| to 2 do
4:       if X_p(i − 1) ≠ cpres(X_p(i)) then
5:         Find a candidate position for task duplication
6:         if there exists unoccupied processor then
7:           nextProc = first unoccupied processor in
8:             fproc(x_{i−1}, 1) to fproc(x_{i−1}, m)
9:         else
10:          nextProc = fproc(x_{i−1}, 1)
11:        end if
12:        Copy current schedule sch to newSch;
13:        In newSch, move X_p(1, 2, ..., i − 1) to nextProc;
14:        Add predecessor trail of X_p(i) to p in newSch;
15:        If makespan(newSch) ≤ makespan(sch)
16:        sch = newSch;
17:      end if
18:    end for
19:    if X_p(1) ≠ entry − node then
20:      Copy current solution schedule to newSch;
21:       Do the same operations from line 9 to line 11 for
          X_p(1);
22:    end if
23:  end for
24: end for
```

3.4 Processor Merging

In the third phase, we try to merge clusters to see if the makespan could be further reduced. Such kind of process is useful, as shown in the DAG example of Fig. 1a. There are two available processors, where the computation cost is 1 on p_1 and 1000 on p_2 for all the tasks. It is obvious that the optimal solution should be scheduling tasks on processor p_1 and no tasks on processor p_2. However, TANH does not output this optimal schedule, as any processor can only hold either task 2 or task 3. This example suggests that when processor model is

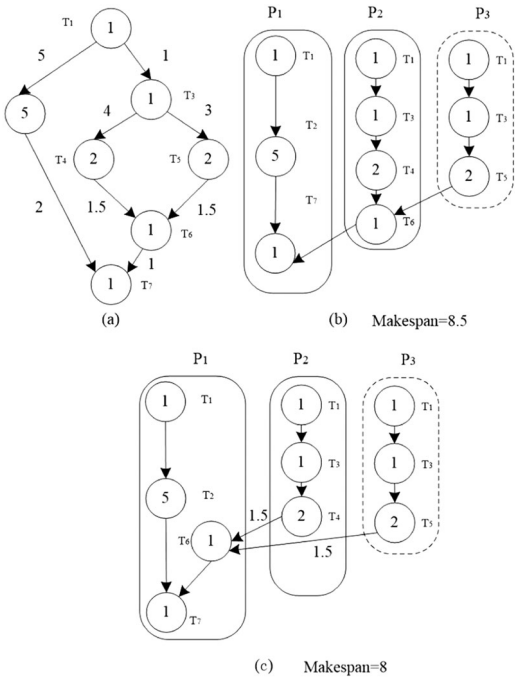


Fig. 2. The EZ benchmark and the TDCA schedule.

strongly heterogeneous, TANH is unlikely to produce high quality solutions.

Consider two candidate methods for the merging phase. Suppose x is the last task processed on processor p and $p = fproc(x, r)$. One way is trying to merge $fproc(x, r)$ with $fproc(x, 1), fproc(x, 2), \ldots, fproc(x, r-1)$ respectively and choose the best among them involving the original schedule. The other way is only trying to combine $fproc(x, 1)$ and $fproc(x, r)$. The former is certainly no worse than the latter, but it takes much longer time.

Therefore, we adopt the second merging strategy: enumerate each processor and consider the last task x assigned on this processor, try to merge with $fproc(x, 1)$. As there still exist replication chains similar to that in the duplication phase, we repeat the above merging procedure for K times. The algorithm for the processor merging is as follows:

Algorithm 3. Processor-Merging Procedure

```
repeat
    for every processor p from 1 to m do
        Copy the current schedule sch to newSch;
        x = the last task on processor p;
        Merge tasks on j to processor fproc(x, 1) in newSch;
        if makespan(newSch) < makespan(sch)
        sch = newSch;
    end for
until the above for-loop for K times
```

3.5 Task Insertion

Finally, we adopt a strategy of task insertion to further reduce the makespan:

For each edge $\langle i, j \rangle$ where task i and j are assigned to different processors p_i and p_j, we try to insert i before j on p_j if this insertion makes j start earlier. In addition, we remove i (except the exit-node) on p_i if i has no outgoing edges on p_i.

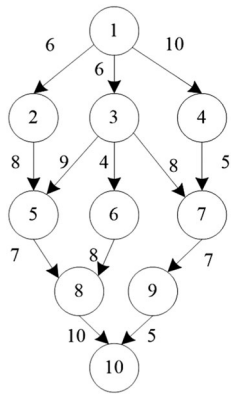


Fig. 3. A DAG example to show the running trace.

To show the necessity of this phase, we use an example of a classic homogeneous task scheduling benchmark, denoted as EZ [11], as shown in Fig. 2a. The computation cost of each task is shown in the node and the edges represent communication costs. We have tasks T_1 to T_7 with task computation cost marked in the nodes and communication cost marked on the edges. After the first three phases, the schedule of TDCA is as shown in Fig. 2b. In the task insertion phase, we duplicate T_6 on processor P_2 and insert T_6 between T_2 and T_7 on processor P_1, as shown in Fig. 2c, and output a schedule of makespan 8.0. The optimal makespan of EZ has been 8.5 [11]. We can prove that optimal makespan of EZ is 8. Here all claims are for sufficient homogeneous machines, which is three in the EZ case.

Lemma 1. *The optimal schedule of EZ is of makespan 8.*

Proof. A task can start if all the output data of its predecessors have arrived. In our schedule, T_1, T_2, T_3, T_4, T_5 are all scheduled in their earliest starting time, and $T_1 \rightarrow T_2 \rightarrow T_7$ is the longest path if we do not consider all the communication cost, so the earliest starting time of T_7 is no less than 6, and its earliest completion time is no less than 7. □

If T_1 and T_2 are not assigned to the same processor, then the earliest completion time of T_2 is 11, and the earliest completion time of T_7 will be 12, which is not optimal. Thus in the optimal schedule, T_1 and T_2 must be assigned to the same processor.

If T_2 and T_7 are not assigned to the same processor, then the earliest completion time of T_7 is no less than 9. So in the optimal schedule, T_2 and T_7 must be assigned to the same processor.

Now for T_6, it only has two choices: (1) in the same processor of T_1, T_2 and T_7: then the earliest starting time of T_6 and T_7 are 6 and 8 respectively; (2) not in the same processor of T_1, T_2 and T_7: then the earliest starting time of T_6 is 5.5, and the earliest completion time of T_7 is 8.5.

Therefore, the makespan of the optimal schedule is 8.

3.6 Trace of TDCA

This subsection gives an example to illustrate how TDCA works in detail. The example DAG is as shown in Fig. 3 and the runtime of tasks on different processors are described in Table 1. The parameters used by TDCA are described in Table 2. For example:

TABLE 1
Computation Cost for the Nodes in Fig. 3

Node \ Processor	P_1	P_2	P_3	P_4	P_5
1	3	6	5	5	4
2	6	3	5	7	3
3	6	5	7	7	6
4	5	6	4	4	4
5	7	2	4	2	2
6	6	7	4	6	5
7	6	4	5	4	6
8	6	6	5	5	6
9	4	5	3	2	3
10	3	1	7	1	2

$$\begin{aligned} est(8,1) &= max\{min\{ect(5,1), ect(5, fproc(5,1)) + C(5,8)\}, \\ &\qquad min\{ect(6,1), ect(6, fproc(6,1)) + C(6,8)\} \\ &= max\{min\{16, ect(5,5) + 7\}, \\ &\qquad min\{15, ect(6,1) + 8\}\} \\ &= max\{min\{16, 12+7\}, min\{15, 15+8\}\} \\ &= 16. \end{aligned}$$

In order to compute $cpred(10)$, we need to compare node 8 and node 9. Since $fproc(8,1) = 3 \neq fproc(10,1) = 5$, the value for node 8 to be compared is $ect(8, fproc(8,1)) + C(8,10) = ect(8,3) + 10 = 21 + 10 = 31$. Since $fproc(9,1) = 4 \neq fproc(10,1) = 5$, the value of node 9 to be compared is $ect(9, fproc(9,1)) + C(9,10) = ect(9,4) + 5 = 18 + 5 = 23$, which is less than 31. Therefore, $cpred(10) = 8$.

Fig. 4 shows the initial clusters achieved at the first phase of TDCA. We sort all nodes in non-decreasing of their levels. The first cluster starts from the node with the smallest level, task 10, and it chooses processor 5, namely $fproc(10,1)$, to hold the cluster. Now $i = 10$ and $j = cpred(10) = 8$. We need to check whether j satisfies inequality (5). As $ect(8,5) = 21$ is less than $ect(8, fproc(8,1)) + C(8,10) = ect(8,3) + 10 = 31$, inequality (8) holds. By adding $cpred(10) = 8, cpred(8) = 6, cpred(6) = 3$ and $cpred(3) = 1$ to processor 5, we obtain the first cluster. Then task 9 is the first unassigned task in the level list. Task 9 and 7 are added to processor $fproc(9,1) = 4$ as usual. Since $y = cpred(7) = 3$ has been assigned, we need to choose a different predecessor for task 7. Task 4 is the only task that satisfies all conditions since it has not been assigned and $ect(4,4) = 9$ is less than $ect(4, fproc(4,1)) + C(4,7) = 13$.

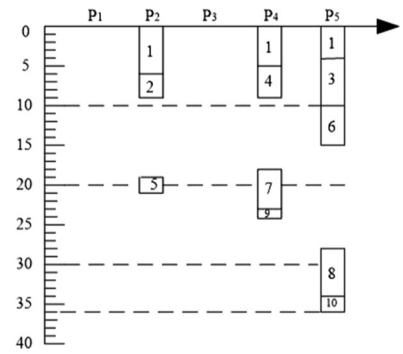


Fig. 4. Initial schedule. (makespan=36).

Even though task 1 has been assigned before, since it is the only predecessor of task 4, we also add task 1 to the cluster. So the second cluster has tasks 9, 7, 4 and 1. Similarly, the third cluster has tasks 5, 2 and 1, and assigned to processor 2.

In the duplication phase, the first position to execute the duplication strategy is between task 5 and task 2 on processor 2. After duplicating the critical predecessor trail of task 5, we have task 5, 3 and 1 on processor 2; task 2 and task 1 are moved to processor 1 since processor 1 is the first available favorite processor for task 2. The new schedule has a makespan of 34, which is shorter than the original one. The next duplication happens between task 7 and task 4 on processor 4. After duplicating the critical predecessor trail of task 7, task 9, 7, 3 and 1 run on processor 4. Task 4 and task 1 are moved to processor 3 since it is the only unoccupied processor. The new schedule still has a makespan of 34. We adopt the new schedule since the makespan does not increase. Though the latter duplication does not reduce the makespan, it may cause other duplications happen and then reduce the makespan. Also, it could be useful in the next phase.

The resulting schedule after the duplication phase considering the duplication chain is shown in Fig. 5. At the final phase of processor merging, as $fproc(2,1) = 5$, we first try to merge processor 1 to processor 5. This attempt is then cancelled as it makes the makespan larger. However, merging processor 2 to processor 5 achieves a good result, and the new schedule has a makespan of 27. One can observe that after the merging, task 5 could use idle time slots on processor 5 and reduce the communication cost with task 5 to 8, and task 8 starts earlier.

TABLE 2
Parameters for the Example DAG in Fig. 3: s for est, c for ect, fp for $fproc$

No.	s_1	s_2	s_3	s_4	s_5	c_1	c_2	c_3	c_4	c_5	fp_1	fp_2	fp_3	fp_4	fp_5	$cpred$	$level$
1	0	0	0	0	0	3	6	5	5	4	1	5	3	4	2	0	65
2	3	6	5	5	4	9	9	10	12	7	5	1	2	3	4	1	52
3	3	6	5	5	4	9	11	12	12	10	1	5	2	3	4	1	53
4	3	6	5	5	4	8	12	9	9	8	1	5	3	4	2	1	41
5	9	11	12	12	10	16	13	16	14	12	5	2	4	1	3	3	37
6	9	11	12	12	10	15	18	16	18	15	1	5	3	2	4	3	38
7	9	12	12	12	10	15	16	17	16	16	1	2	4	5	3	3	30
8	16	18	16	18	15	22	24	21	23	21	3	5	1	4	2	6	23
9	15	16	17	16	16	19	21	20	18	19	4	1	5	3	2	7	17
10	22	24	21	23	21	25	25	28	24	23	5	4	1	2	3	8	7

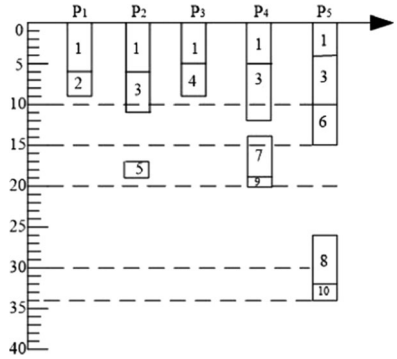


Fig. 5. After duplication. (makespan=34).

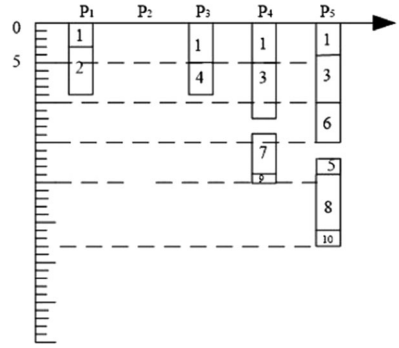


Fig. 6. After merging. (makespan=27).

Since task 9 could complete earlier than the initial schedule by the duplication phase, and task 8 could complete earlier at the processor merging phase, task 10 could start earlier. Therefore, we have a smaller makespan. The duplication chain does not affect at this phase. Then in the task insertion phase, no insertion will be added. The ultimate schedule is shown in Fig. 6. This example demonstrates the optimization of TDCA. By comparison, the schedule of TANH is also shown in Fig. 7, which has a makespan of 31.

3.7 Complexity Analysis

The time complexity of calculating est, $fproc$, and other parameters is $O(me)$, $O(mn \log m)$, and $O(e)$ respectively. So the overall complexity of computing the parameters is $O(me + mn \log m)$.

In the task clustering phase, we need to sort tasks with a complexity of $O(n \log\ n)$. Every node may have to find an unoccupied processor with a complexity of $O(mn)$. This check step involves enumerating all predecessors with a complexity of $O(e)$. So the complexity of task clustering is $O(n \log n + mn + e)$.

In the duplication phase, there are at most $O(n)$ task duplication positions. After each duplication phase, we need to recalculate the makespan, whose complexity is $O(me)$. The complexity of task duplication is at most $O(mne)$.

In the processor merging phase, the complexity of enumerating processors is $O(m)$, and the complexity of recalculating the makespan is still $O(me)$. So the total complexity of the processor merging is $O(m^2e)$.

Finally in the task insertion phase, there are at most $O(n)$ task insertion positions. For each insertion phase, we need to insert some tasks, whose complexity is $O(1)$. So the complexity of task insertion is at most $O(n)$.

To sum up, the overall time complexity of TDCA is $O(mne + m^2e)$. We know that the complexity of TANH is $O(mne)$ if we counts the complexity of recalculating the makespan after each duplication phase. We can find that TDCA has the same complexity with TANH when m is comparable to n.

The space complexity of TDCA is $O(mn)$ as each task can duplicate at most m times.

4 EXPERIMENTS AND ANALYSIS

4.1 DAG Graph Generator

As we did not find enough DAG benchmarks in the literature, we systematically generate random DAG graphs with various structures to compare TDCA with several state-of-the-art existing algorithms, and analyze the task duplication paradigm. Specifically, we vary several key parameters to investigate their impact on the scheduling algorithms, namely the number of tasks n, the communication-computing cost ratio CCR, the heterogeneity parameter h, and the depth of the DAG L. We generate random graphs with the following steps.

1) *Generate nodes in layers*. The generated DAG consists of L layers and nodes are assigned to layers. Nodes in one layer only have edges pointing to the nodes in the next layer. The first layer only contains one node, the entry-node. The last level only contains one node, the exit-node. The remaining $n - 2$ nodes are evenly distributed to the other $L - 2$ layers.
2) *Generate edges between nodes in adjacency layers*. We ensure that each node has at least one predecessor node and one successive node.
3) *Generate computation costs*. We use the heterogeneity parameter h to control the variance of the computing costs of a task on different processors. For task i, let $T(i, p)$ represent its computation cost when assigned to processor p. $T(i, p)$ follows a uniform distribution and has a pre-determined mean value.
4) *Generate communication costs*. The communication-computing cost ratio, CCR, is an important parameter that greatly influences the performance of different scheduling algorithms. CCR determines the ration between the pre-determined mean value of computing costs and the pre-determined mean value of communication costs.

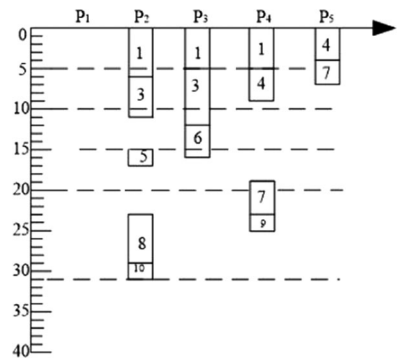


Fig. 7. Schedule of TANH. (makespan=31).

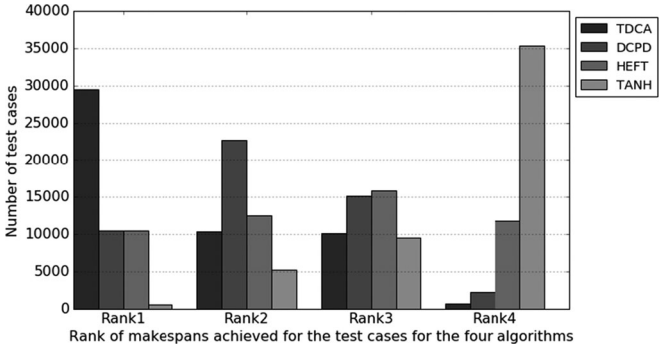


Fig. 8. Comparison on the number of test cases in different rankings. TDCA has the shortest makespan for most test cases (58.14 percent), as shown in Rank1; and TDCA outputs the longest makespan on very little small portion of the test cases (0.014 percent), as shown in Rank4.

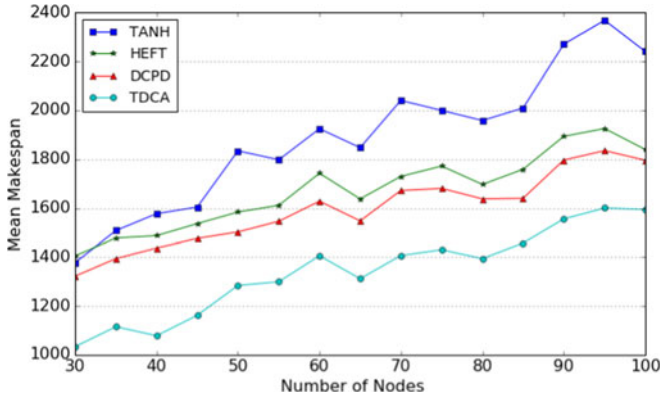


Fig. 9. Comparison of makespan on number of nodes. TDCA yields the shortest makespan.

4.2 Comprehensive Comparison

We conduct two experiments to compare the performance of TDCA, TANH and HEFT and analyze how the parameters affect their performance. In the first experiment, we compare the makespans of the schedules produced by TDCA, TANH and HEFT. We generate test graphs by using the following parameters:

1) $n \in$ {30,35,40,45,50,55,60,65,70,75,80,85,90,95,100}
2) $L \in$ {3,4,5,6,7,8,9,10,11,12,13,14,15,16,17}
3) $CCR \in$ {0.1,0.3,0.5,0.7,1,1.2,1.5,1.8,2,2.5,3,5,7,10,15}
4) $h \in$ {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1,1.1,1.2,1.3,1.4}

The number of processors m is set to be $maxWidth(G)$, the width of the DAG, defined to be the max value among the numbers of edges between any two adjacent layers. For example for the DAG in Fig. 3, the maxWidth is 5. This definition is adequate for any algorithm to achieve good performance. Note that it is OK to use larger or smaller values to define the number of processors for our DAG generator. We generate 50,625 test cases, test TDCA, DCPD, TANH, HEFT on these test cases, and rank the algorithms basing on their makespan of their schedules. Fig. 8 shows the ranking result, from which we see that TDCA produces the best schedule in 58.14% (29,434 out of 50,625) test cases, the following schedule in 20.56% (10,410 out of 50,625). By comparison, DCPD is in rank 2 for most test cases, and HEFT is in rank 3 for most cases. The result shows that TDCA outperforms DCPD, TANH and HEFT on most test cases.

In the second experiment, we investigate the impact of key parameters on the performance of the algorithms. Note that the parameter space is too large for us to exhaustively explore their effects. To simplify the analysis, we assume that the parameters affect the performance of the algorithms independently. The standard parameter configurations are shown in Table 3. By varying one parameter while fixing the others in the baseline configuration, we can investigate the influence of each parameter. As the specific weights and the graph topological structure are randomly generated, one algorithm usually produces different schedules with different makespans under the same configuration, which impedes us observing the effect of a parameter. We reduce this variance by repeatedly generating test cases and presenting the mean values.

We first investigate the impact for the number of nodes n, as shown in Fig. 9. All algorithms scale linearly with the number of nodes and TDCA always produces the smallest mean makespan.

The impact of CCR is shown in Fig. 10. When CCR is small, the communication cost between tasks can be negligible and task computation cost plays a key role in the resulting makespans. When CCR is very large, communication cost mainly influences the makespans. Every algorithm produces larger makespan when CCR increases since communication becomes more expensive. When CCR is small ranging from 0.1 to 0.3, HEFT, DCPD and TDCA perform similarly. When CCR is ranging from 0.3 to 13, TDCA outperforms the other three. DCPD is slightly better than HEFT, and they are better than TANH. In general, TDCA is the best performing algorithm.

The heterogeneity parameter h represents the heterogeneity of computing power among processors. Value 0 is for homogeneous computing environment where a task has the same execution time on all processors. The larger the value of h is, the more heterogeneous the system is. The results presented in Fig. 11 show the mean makespan of different algorithms on different values of parameter h. In all ranges,

TABLE 3
Baseline Configuration for the Parameters

n	L	m	CCR	h
50	7	100	2.0	0.7

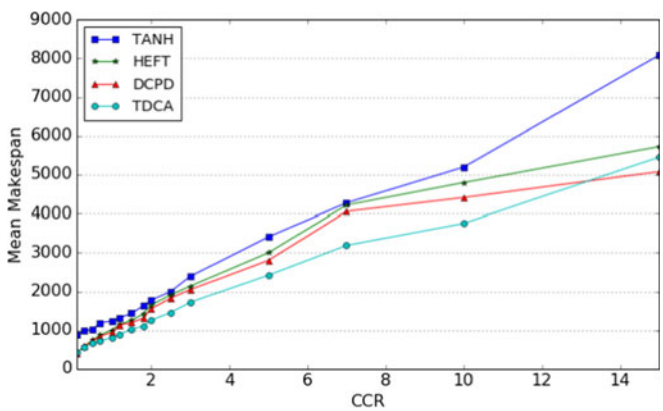


Fig. 10. Comparison of makespan on CCR. TDCA yields the shortest makespan in most cases.

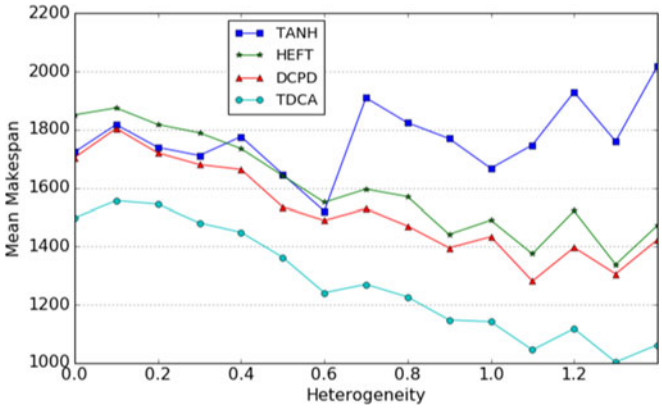


Fig. 11. Comparison of makespan on heterogeneity. TDCA yields the shortest makespan.

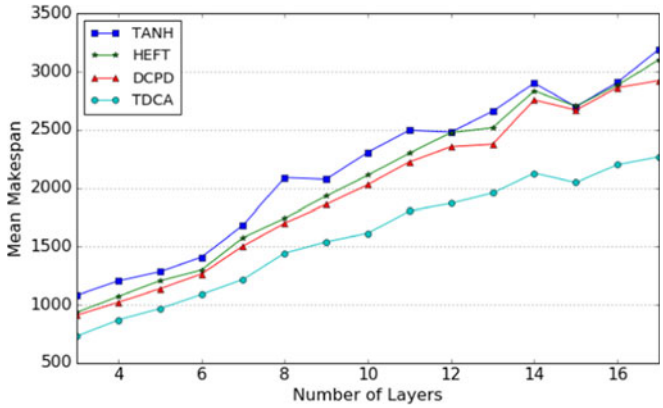


Fig. 12. Comparison of makespan on number of layers. TDCA yields the shortest makespan.

TDCA produces the shortest schedules. A larger h indicates that every task has very different computation costs on different processors. In the extreme case, the task takes very little time on one processor but takes very long time on other processors, the algorithms then know how to effectively allocate the tasks to the proper processors to reach a small makespan. We observe that TDCA, DCPD, HEFT follow the similar decreasing trend for stronger heterogeneity. TANH is less stable with a larger variance and it even increases the makespan after $h = 0.6$. This result indicates that TANH is not good for strong heterogeneity.

The last parameter is the number of layer L. The results shown in Fig. 12 show the mean makespan of different algorithms on different number of layers. Makespans increase linearly with the number of layers, which meets our expectation since the large number of layer indicates low parallelism.

In general, TDCA generates the best schedule on all test cases. Among various parameters of the graphs, CCR has the greatest impact on the performance of different algorithms. Other parameters can also influence makespans, but they do not change the ranks of algorithms.

4.3 Analysis on the Duplication

To further analyze the task duplication process and determine the best value for the number of iterations (value K in Algorithm 2), we modify the task duplication phase of TDCA by iteratively searching for candidate duplication positions. At each round of the iteration, we enumerate processors from 1 to m and check every candidate position for the duplication. Duplications are kept if they do not grow the makespan. If in a certain round of iteration, no task duplication is adopted, then we terminate the duplication process.

To determine the value of K, we count how many rounds of iterations are needed to exhaust all the candidate task duplication positions. The results are shown in Fig. 13. One round of iteration means that no task duplication can improve the original schedule and two rounds of iterations indicate that one round of iteration can locate all beneficial task duplication candidate positions. One and two rounds of iterations consist of more than half of test cases when the number of nodes is small. Their ratio decreases dramatically when the number of node increases. The percentage is 55.88 percent when the number of nodes is 30 and it is 12.86 percent when there are 100 nodes. This result demonstrates that there are many task replication chains and other algorithms miss these opportunities.

When the number of nodes is around 100, four rounds of iterations are sufficient for 97.48 percent test cases. Since the ratio of four rounds of iterations is still rising, it is reasonable to predict that $K = 4$ would be enough for even larger graphs. On the other hand, it is important to see how much improvement we can achieve when conducting multiple rounds of iterations.

The results demonstrate that most improvement is gained at the first round of iteration, which is about 12 percent. The second round of iteration can further achieve about 2 percent improvement. The following rounds of iterations have less than 1 percent improvement. It can be explained by the 80-20 rule that roughly 80 percent of the effects come from 20 percent of the causes. Therefore, we have to pay more time to gain more improvement. There is an obvious tradeoff between complexity of the algorithm and the quality of the schedule. Generally speaking, the existence of replication chains in the phase of task duplication truly degenerate the solution qualities of TANH since there is a trend that more rounds of iterations are required to exhaust all the task duplication candidate positions with the growth of the node numbers. Our analysis shows that the more rounds of iterations are used, the more valid task duplication positions we can find. However, the more rounds of iterations are used, the less gain we can get from each round of iteration. We check out that four times of iteration can find most valid task duplication candidate positions and it can achieve a reasonable improvement at each

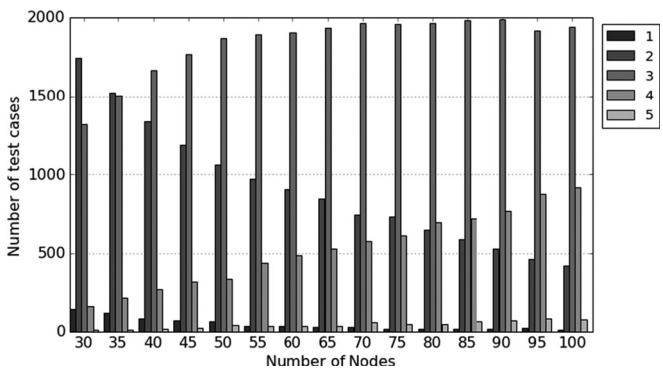


Fig. 13. Number of test cases that achieve local optimum at different rounds of the task duplication.

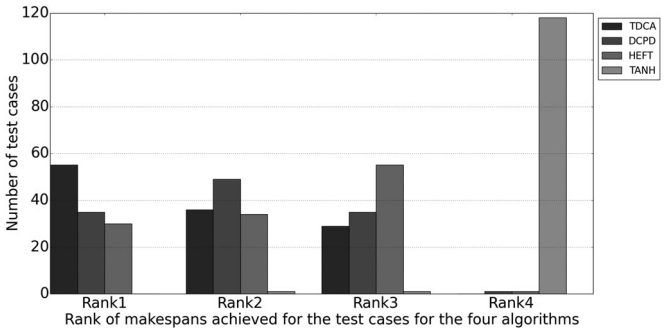


Fig. 14. Comparison on the number of test cases in different ranking with large DAGs for the four algorithms. TDCA has the shortest makespan among the four algorithms for most test cases (45.83%), as shown in Rank1; and TDCA never produces the longest makespan, shown in Rank4.

round of iteration. Therefore, TDCA iterates the process four times to get shorter solutions.

4.4 Further Test on Large Scale DAGs

To compare the algorithms in a large scale, we generate 120 test cases for large DAGs using the following parameters:

1) number of tasks $n \in \{2000, 2300, 2700, 3000\}$
2) depth of the DAG $L \in \{200, 300\}$
3) commu-compu ratio $CCR \in \{0.3, 0.7, 1.2, 1.8, 2.0\}$
4) heterogeneity parameter $h \in \{0, 0.5, 1.0\}$

Here "commu-compu ratio" indicates the communication-computing cost ratio of the DAGs. In this experiment, we set $m = 100$ as it is important to test the scheduling qualities when the number of processors is bounded to a reasonable value. Fig. 14 illustrates the rankings. Among the 120 test cases, TDCA ranks the first in 45.83% (55 out of 120) test cases; DCPD ranks the first in 29.17% (35 out of 120) test cases; HEFT ranks the first in 25.00% (30 out of 120) test cases. Unfortunately TANH never ranks the first, and we notice that the makespans of schedules generated by TANH are significantly longer than that of the other three algorithms.

It shows that TANH may not properly handle the cases when the number of tasks is much larger than the number of processors. This experiment demonstrates that TDCA makes substantive progress based on TANH, and TDCA is also competitive compared to state-of-the-art algorithms for large scale DAG scheduling problems.

5 Conclusion

We propose a novel algorithm called TDCA for the DAG task scheduling problem in the heterogeneous distributed environment. The proposed algorithm utilizes the duplication task more thoroughly involving parameter calculation, task duplication, and task merging. We redefine several key parameters est, $cpred$ by considering the most favorite processor for each task, the processor that the task has the earliest completion time if the task is assigned on the processor. When calculating est, we ignore the communication cost if the task and its predecessor are on the same processor, which yields a more accurate est. The meanings of ect and $fproc$ are changed accordingly. These critical concepts have been improved to build the initial clusters.

We consider both the duplication phase and the merging phase such that the makespan could be further shortened. During the duplication phase, we design a method to determine the order of finding the candidate duplication positions. By analyzing the effect of task duplication, we observe that chain reactions exist in the process of task duplication. After one round of scanning all duplication candidates, we can classify all duplication candidates into effective candidates, which will be applied to improve the overall makespan, and ineffective candidates, which will be put aside temporarily. In the next round of task duplication, these ineffective candidates can become effective because the current schedule has changes from the starting of the previous round. TDCA then fully utilizes this chain reaction phenomenon, and improves the scheduling performance.

Several comprehensive comparison experiments are conducted to demonstrate that TDCA can significantly improve the scheduling quality and explicitly outperform the baseline algorithms without increasing the computation complexity.

Acknowledgments

This work was supported by National Natural Science Foundation of China (Grant No. 61472147, 61602196, and 61401169).

References

[1] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.

[2] S. Hong, T. Chantem, and X. S. Hu, "Local-deadline assignment for distributed real-time systems," *IEEE Trans. Comput.*, vol. 64, no. 7, pp. 1983–1997, Jul. 2015.

[3] M. Hu and B. Veeravalli, "Requirement-aware scheduling of bag-of-tasks applications on grids with dynamic resilience," *IEEE Trans. Comput.*, vol. 62, no. 10, pp. 2108–2114, Oct. 2013.

[4] S. Baskiyar and C. Dickinson, "Scheduling directed a-cyclic task graphs on a bounded set of heterogeneous processors using task duplication," *J. Parallel Distrib. Comput.*, vol. 65, no. 8, pp. 911–921, Aug. 2005.

[5] A. Talukder, M. Kirley, and R. Buyya, "Multiobjective differential evolution for scheduling workflow applications on global grids," *Concurrency Comput.: Practice Exp.*, vol. 21, no. 13, pp. 1742–1756, 2009.

[6] K. Deng, K. Ren, S. Liu, and J. Song, "DAG scheduling for heterogeneous systems using biogeography-based optimization," in *Proc. IEEE 21st Int. Conf. Parallel Distrib. Syst.*, Dec. 2015, pp. 708–716.

[7] I. Gupta, M. S. Kumar, and P. K. Jana, "Task duplication-based workflow scheduling for heterogeneous cloud environment," in *Proc. 9th Int. Conf. Contemporary Comput.*, Aug. 2016, pp. 1–7.

[8] J. Singh, B. Mangipudi, S. Betha, and N. Auluck, "Restricted duplication based milp formulation for scheduling task graphs on unrelated parallel machines," presented at the *5th Int. Symp. Parallel Architectures Algorithms Program.*, Taipei, Taiwan, Dec. 2012.

[9] M. I. Daoud and N. Kharma, "A high performance algorithm for static task scheduling in heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 68, no. 4, pp. 399–409, 2008.

[10] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999.

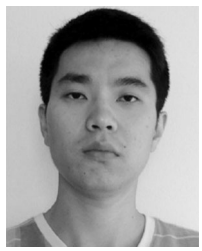
[11] P. Chowdhury and C. Chakrabarti, "Static task-scheduling algorithms for battery-powered dvs systems," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 13, no. 2, pp. 226–237, Feb. 2005. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2004.840771

[12] C. L. McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle, "A comparison of heuristics for scheduling DAGs on multiprocessors," in *Proc. 8th Int. Parallel Process. Symp.*, Apr. 1994, pp. 446–451.

[13] M. A. Palis, J.-C. Liou, and D. S. L. Wei, "Task clustering and scheduling for distributed memory parallel architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 1, pp. 46–55, Jan. 1996.

[14] G.-L. Park, B. Shirazi, and J. Marquis, "DFRN: A new approach for duplication based scheduling for distributed memory multiprocessor systems," in *Proc. 11th Int. Parallel Process. Symp.*, Apr. 1997, pp. 157–166.
[15] S. Darbha and D. P. Agrawal, "Optimal scheduling algorithm for distributed-memory machines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 1, pp. 87–95, Jan. 1998.
[16] A. Radulescu and A. J. C. van Gemund, "Low-cost task scheduling for distributed-memory machines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 6, pp. 648–658, Jun. 2002. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2002.1011417
[17] R. Sakellariou and H. Zhao, "A hybrid heuristic for dag scheduling on heterogeneous systems," in *Proc. 18th Int. Parallel Distrib. Process. Symp.*, 2004, Art. no. 111.
[18] S. Ranaweera and D. P. Agrawal, "A task duplication based scheduling algorithm for heterogeneous systems," in *Proc. 14th Int. Parallel Distrib. Process. Symp.*, May 2000, pp. 445–450.
[19] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.
[20] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 682–694, Mar. 2014.
[21] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra, "Hierarchical DAG scheduling for hybrid distributed systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 156–165.
[22] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang, "Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing," in *Proc. Int. Conf. Supercomputing*, 2016, Art. no. 20.
[23] H. Zhao and R. Sakellariou, "Scheduling multiple dags onto heterogeneous systems," in *Proc. Parallel Distrib. Process. Symp.*, 2006, Art. no. 14 pp.
[24] K. Li, X. Tang, B. Veeravalli, and K. Li, "Scheduling precedence constrained stochastic tasks on heterogeneous cluster systems," *IEEE Trans. Comput.*, vol. 64, no. 1, pp. 191–204, Jan. 2015.
[25] F. A. Omara and M. M. Arafa, "Genetic algorithms for task scheduling problem," *J. Parallel Distrib. Comput.*, vol. 70, no. 1, pp. 13–22, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2009.09.009
[26] G. Q. Liu, K. L. Poh, and M. Xie, "Iterative list scheduling for heterogeneous computing," *J. Parallel Distrib. Comput.*, vol. 65, no. 5, pp. 654–665, May 2005.
[27] X. Tang, K. Li, G. Liao, and R. Li, "List scheduling with duplication for heterogeneous computing systems," *J. Parallel Distrib. Comput.*, vol. 70, no. 4, pp. 323–329, Apr. 2010.
[28] J. Ali and R. Z. Khan, "Optimal task partitioning model in distributed heterogeneous parallel computing environment," *Int. J. Adv. Inf. Technol.*, vol. 2, no. 6, 2012, Art. no. 13.
[29] S. Kim and J. Browne, "General approach to mapping of parallel computations upon multiprocessor architectures," in *General Approach to Mapping Of Parallel Computations Upon Multiprocessor Architectures*, 1998, vol. 3, pp. 1–8.
[30] S. Gupta, R. Rajak, G. K. Singh, and S. Jain, "Review of task duplication based (tdb) scheduling algorithms," *Smart CR*, vol. 5, pp. 67–75, 2015.
[31] K. He and Y. Zhao, "A new task duplication based multitask scheduling method," in *Proc. 5th Int. Conf. Grid Cooperat. Comput.*, Oct. 2006, pp. 221–227.
[32] K. Shin, M. Cha, M. Jang, J. Jung, W. Yoon, and S. Choi, "Task scheduling algorithm using minimized duplications in homogeneous systems," *J. Parallel Distrib. Comput.*, vol. 68, no. 8, pp. 1146–1156, 2008.
[33] C.-H. Liu, C.-F. Li, K.-C. Lai, and C.-C. Wu, "A dynamic critical path duplication task scheduling algorithm for distributed heterogeneous computing systems," in *Proc. 12th Int. Conf. Parallel Distrib. Syst.*, vol. 1, Jul. 2006, Art. no. 8.
[34] R. Bajaj and D. P. Agrawal, "Improving scheduling of tasks in a heterogeneous environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 2, pp. 107–118, Feb. 2004.
[35] G. Xie, R. Li, X. Xiao, and Y. Chen, "A high-performance dag task scheduling algorithm for heterogeneous networked embedded systems," in *Proc. IEEE 28th Int. Conf. Adv. Inf. Netwo. Appl.*, 2014, pp. 1011–1016. [Online]. Available: http://dx.doi.org/10.1109/AINA.2014.123

Kun He received the BS degree from the Department of Physics from Wuhan University, in 1993, Wuhan, P.R. China, the MS degree from the Department of Computer Science, Huazhong Normal University, Wuhan, P.R. China, in 2002, and the PhD degree from the Department of Automatic Control from HUST, in 2006, Wuhan, P.R China. She is currently a professor with the Department of Computer Science, Huazhong University of Science and Technology (HUST), Wuhan, P.R. China; and a Mary Shepard B. Upson Visiting Professor for the 2016-2017 Academic year in Engineering, Cornell University, New York. Her research interest includes algorithm design and analysis, distributed computation, machine learning.



Xiaozhu Meng received the BS degree from the Department of Computer Science, HUST, in 2011, Wuhan, P.R. China. He is a currently working toward the PhD degree in Computer Science Department, University of Wisconsin Madison, Wisconsin. His current research interest includes parallel and distributed system.

Zhizhou Pan received the MS degree in computer science from HUST, in 2015, Wuhan, P.R. China. He is currently a senior technical member for mobile computing with Tencent Company, Shenzhen, P.R. China. His current research interest includes parallel and distributed system, mobile computing.

Ling Yuan received the BS and MS degrees in computer science from Wuhan University, Wuhan, China, in 1997 and 2002, respectively, and the PhD degree from the Department of Computer Science, National University of Singapore, in 2008. She is currently an associate professor with the School of Computer Science and Technology, HUST, Wuhan, P.R. China. Her research interest includes parallel and distributed system, and software engineering.

Pan Zhou (S'07, M'14) received the BS degree in the advanced class from HUST, the MS degree from the Department of Electronics and Information Engineering, HUST, Wuhan, China, in 2006 and 2008, respectively, and the PhD degree from the School of Electrical and Computer Engineering, Georgia Institute of Technology (Georgia Tech), in 2011, Atlanta. He received honorary degree in his bachelor and merit research award of HUST in his master study. He is currently an associate professor with the School of Electronic Information and Communications, Wuhan, P.R. China. He was a senior technical member with Oracle Inc, America during 2011 to 2013, and worked on Hadoop and distributed storage system for big data analytics at Oracle Cloud Platform. His current research interest includes big data analytics and machine learning, security and privacy, and information networks. He is a member of the IEEE.