Adaptive Scheduling of Task Graphs with Dynamic Resilience

Menglan Hu, Jun Luo, *Member, IEEE*, Yang Wang, and Bharadwaj Veeravalli, *Senior Member, IEEE*

Abstract—This paper studies a scheduling problem of task graphs on a non-dedicated networked computing platform. The networked platform is characterized by a set of fully connected processors such as a multiprocessor system that can be shared by multiple tasks. Therefore, the computation and communication capacities of the computing platform dynamically fluctuate. To deal with this fluctuations for high performance task graph computing, we propose an online dynamic resilience scheduling algorithm called *Adaptive Scheduling Algorithm* (ASA) that bears certain distinct features compared to existing algorithms. First, the proposed algorithm deliberately assigns tasks to idle processors in multiple rounds to prevent any unfavorable decisions and also to avoid inefficient assignments of certain key tasks to slow processors. Second, the algorithm adopts task duplication as an attempt to minimize serious increase of schedule length due to unexpected processor slowdown. Finally, a look-ahead message transmission policy is applied to save communication time and further improve the overall performance. Performance evaluation results are presented to demonstrate the effectiveness and competitiveness of our approaches when compared with the existing algorithms.

Index Terms—Dynamic algorithm, dynamic resilience, multiprocessor scheduling, task graphs, task duplication.

1 INTRODUCTION

Efficient scheduling of the tasks in a parallel application onto multiprocessors is a critical issue for achieving high performance. In the general form of a parallel task scheduling problem, an application is represented by a directed acyclic graph (DAG), where vertices represent tasks and edges represent inter-task communications. The objective is to schedule tasks onto a set of processors such that the makespan (i.e., overall execution time) of the DAG is a minimum. The scheduling problem is known as NP-hard [1].

In this paper, we investigate a dynamic scheduling problem of performing a task graph on a non-dedicated multiprocessor. Nowadays computer systems are customarily multiprocessor/multi-core systems with multi-tasking functionality. Such computers simultaneously run multiple tasks (i.e,. processes) in multiple processors/cores. Each single processor/core can also perform multiple processes concurrently by switching its time slices (i.e., CPU cycles) between different tasks. Because of the time sharing, CPU cycles (i.e., CPU speed) available to one task may dynamically fluctuate throughout the execution of the task. Additionally, multiple tasks may also contend for other type of resources such as memory and communication resources. Therefore, in such a non-dedicated computer system, available computation speed for each given task may dynamically vary with time. For instance, the available speed may drop down when multiple other tasks are launched in the system and may rebound after these tasks leave the system.

The dynamic scheduling problem in multiprocessors has wide applicability. For example, computer clusters (in a department or a laboratory) can simultaneously run many tasks which may belong to one or multiple users. Accordingly, for each task, its available CPU cycles may fluctuate over time. Grid is another kind of shared systems. One typical example is SETI@home [20], which exploits donated (thus unguaranteed and fluctuating) computer power across the world to perform a huge amount of computations. Besides, even in cloud platforms wherein virtual machines (VMs) are dedicated to one user who leases them, the user may simultaneously run multiple tasks in one VM, making the VMs a shared and dynamic computing system.

Since most practical multiprocessor systems are shared and dynamic, it is highly desirable to investigate the scheduling problem for non-dedicated (i.e., shared) multiprocessors. Although the classical task graph scheduling problem was extensively investigated by prior work, most of the work (e.g., [10], [7], [8], [9]) only focused on the static scheduling problem wherein processor speeds are assumed to be fixed. However, when considering dynamic scenarios, their heuristics often run into the so-called *pre-committing* problem. That is, their heuristics statically determine the global schedule of all tasks before the execution. In this case, it is very common that the available capacity of the assigned processor for the task unexpectedly slows down due to the contention of other tasks. The slowed processor will delay the execution of the task and may also impede the execution of the whole task graph if it is on the critical path. In other words, the scheduling decisions are made too early and may become inefficient as the available processor capacity dynamically varies.

- *M. Hu is with the School of Electronic Information and Communications, Huazhong University of Science and Technology, Wuhan, China 430074. E-mail: humenglan@hust.edu.cn.*
- *J. Luo is with the School of Computer Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore 639798. E-mail: junluo@ntu.edu.sg.*
- *Y. Wang is with Shenzhen Institute of Advanced Technology, Chiniese Academy of Science, E-mail: yang.wang1@siat.ac.cn*
- *B. Veeravalli is with the Department of Electrical and Computer Engineering, National University of Singapore, 4 Engineering Drive 3, Singapore 117576. E-mail: elebv@nus.edu.sg.*

Although online algorithms [3], [4], [5] can alleviate the pre-committing problem to some degree, they receive only limited attention owing to the dynamism and complexity of the task graphs and computing resources. Moreover, these studies still neglected the dynamic properties of underlying compute resources. We will present a detailed review on related online scheduling algorithms in the next section. Such algorithms can be categorized into two types. The first type (e.g., [3]) only considers scheduling tasks to idle processors. In this case, tasks may be assigned to idle but slow processors, thereby degrading the overall performance. The second type considers both idle and busy processors for assigning tasks. This type of strategy may allocate tasks to fast processors which are still processing previously assigned tasks. This is again a pre-committing situation. Once the fast processors unexpectedly decelerates, the assignment becomes inefficient. Unfortunately, these algorithms cannot effectively react to and fix inefficient scheduling decisions in presence of inaccurate task profiling and fluctuating computing capacities.

As explained above, the scheduling of task graphs in shared multiprocessors is very difficult. Even worse, in practice task profiling techniques hardly estimate perfectly accurate information on the actual workloads of the tasks. Inaccurate estimate information may further deteriorate the performance of a schedule in actual execution. When we consider task graphs, the scheduling problem becomes even more challenging as unexpectedly slowed processors and inaccurate predictions on both tasks and processors may delay the completion of some tasks, which may further hinder the execution of all successors of the delayed tasks and thereby defer the overall progress.

Motivated by these needs and challenges, this paper investigates the scheduling of a task graph on a non-dedicated multiprocessor. We contribute a dynamic task graph scheduling algorithm called *Adaptive Scheduling Algorithm* (ASA) that realistically deals with the dynamic properties of multiprocessor platforms in several ways. First, the proposed algorithm assigns tasks to idle processors in multiple rounds with a tentative scheduling strategy to prevent inefficient pre-committing decisions and avoid unfavorable assignment of key tasks to slow processors. Second, the algorithm applies task duplication to prevent delayed task completion due to unexpected processor slowdown. Third, a look-ahead message transmission policy is applied to save communication time. In short, the essence of the paper is to trade algorithm complexity and computing resources for better performance.

The remainder of this paper is organized as follows. Section 2 discusses the related work. Section 3 introduces mathematical models, assumptions, and problem formulation. Section 4 describes the proposed algorithms in great detail. Section 5 presents simulation results to evaluate the algorithm, with conclusions following in Section 6.

2 Related Work

The problem of task graph scheduling has been extensively studied in past decades and many heuristic algorithms were proposed. The heuristics are classified into a variety of categories such as *list scheduling*, *clustering*, and *duplication-based* algorithms, which are mainly for static scheduling on homogeneous systems. A list scheduling heuristic maintains a list of all tasks of a given graph according to their priorities. It has two phases: the task selection phase for selecting the highest-priority ready task and the processor selection phase for selecting a suitable processor that minimizes a predefined cost function. Some of the examples are the Modified Critical Path (MCP) [9], Earliest Task First (ETF) [7], Dynamic Critical Path (DCP) [8], and Heterogeneous Earliest-Finish-Time (HEFT) [10] algorithms. Kwok and Ahmed surveyed a number of list scheduling heuristics in [2]. As list scheduling approaches can provide high performance at a low cost, our paper presents algorithms based on list scheduling techniques. Another type of heuristic is clustering [11], [9]. In this category, tasks are pre-clustered before allocation begins to reduce the problem size. Task clusters (instead of individual tasks) are then assigned to individual processors. Algorithms in this category are typically static and thus inappropriate to dynamic cases. Duplication-based scheduling algorithms [13], [14], [15], duplicate tasks across multiple processors to reduce communication overheads. The algorithms in this group are usually for an unbounded number of homogeneous processors and they have much higher complexity values than the algorithms in other groups.

In contrast to duplication-based algorithms, some grid scheduling algorithms [6], [12] apply task duplication to prevent delayed task completion. These heuristics require no performance prediction information of underlying resources and are thereby called knowledge-free approaches. Another type of grid scheduling approaches is so-called knowledge-based, which customarily assumes perfect performance prediction information on resources and tasks. This intuition is similar to the ideas adopted in prior grid scheduling heuristics. Some well-known knowledge-based heuristics include Max-Min, Min-Min, Sufferage [17], XSufferage [18], and Storage Affinity (SA) [19]. Although these algorithms are efficient when the prediction information is available, yet in practice such information is not always available. Since both knowledge-based and knowledge-free approaches are useful, in this paper we borrow the intuitions from both of them.

The above heuristics are mainly designed for static scheduling and often run into the problem of pre-committing the schedule to the processors when the scenario extends to dynamic situations. A task then can only be executed on the assigned processors even if a favorable rescheduling is possible on another processor. Though online scheduling schemes do not have this constraint, they have not been studied extensively given the dynamism and complexity of the computer resources and applications. Feldmann et al. [4] proposed a method for online scheduling of parallel applications assuming that execution time is unknown beforehand. In contrast, Choudhury et al. [5] designed a hybrid offline and online approach for dynamic task graphs without communication costs which shows improvement over pure static schedulers. [3] presented an online scheduling algorithm for conditional task graphs with communication contention in multiprocessors. Nevertheless, these studies neglected the dynamism in task properties (e.g., inaccurately estimated task sizes) and underlying computing resources. As a result, these proposed algorithms cannot

Parameter	Definition
C_k	estimated computation speed of P_k
e_{ij}	edge linking two nodes n_i and n_j
n_i	i-th node in the task graph ($i = 1, 2, \cdots, M$)
P_k	k-th processor ($k = 1, 2, \cdots, K$)
R_i	number of actual instances of n_i concurrently being computed
R_i^v	number of virtual instances of n_i
$rank(n_i)$	rank of n_i
t_i^{TF}	temporary finish time of n_i
t_k^{TA}	temporary available time of P_k
t_k^{EA}	estimated available time of P_k for its running task
S^R	set containing ready nodes
w_i	workload of n_i
w_{ij}	estimated time cost for transferring e_{ij}
$\bar{w_i}$	average execution time cost of n_i

TABLE 1
Notations and Terminology

effectively react and fix inefficient scheduling decisions upon the actual task execution in presence of inaccurate predictions on tasks and fluctuating available computing capacities.

3 Problem Formulation

We consider scheduling a task graph on a multiprocessor. Let $P = \{P_1, P_2, \cdots, P_K\}$ denote the set of processors. Since the processors are shared among multiple users, they could not be dedicated to a single task graph. As a consequence, the computation speed of the processors will fluctuate over time. We assume that prediction information on the compute resources is available to the scheduler owing to prediction mechanisms proposed in prior works [16]. However, the prediction information may be inaccurate as the capacities of the processors may dynamically fluctuate along the time axis. The estimated computation speed of processor P_k ($j = 1, 2, ..., K$) is denoted as C_k.

An application g can be modeled by a directed acyclic graph (DAG) $G = (V, E)$, where V is the set of vertices (i.e., tasks) and E is the set of edges. A vertex corresponds to a task which is a set of instructions that must be executed serially on the same processor. Terms *task* and *node* are interchangeably used thereafter in this paper. Associated with n_i ($i = 1, 2, \cdots, M$) is its workload (required amount of computations) w_i. Before scheduling, each task is labeled with the estimated execution time cost $\bar{w_i}$, defined as:

$$\bar{w_i} = \frac{K w_i}{\sum\limits_{k=1}^{K} C_k} \tag{1}$$

An edge e_{ij} linking two nodes n_i and n_j specifies the communication and precedence between the two nodes. That is, n_i should complete its execution before n_j starts. Also, let w_{ij} be estimated time cost for transferring e_{ij}. Notice that both w_i and w_{ij} known to the scheduler may be inaccurate and the proposed algorithm can tolerate such inaccuracy to some degree. The source node n_i and the destination node n_j of the edge e_{ij} are called the *parent* node and the *child* node, respectively. A node which has no parent is called an entry node while a node has no child is called an exit node.

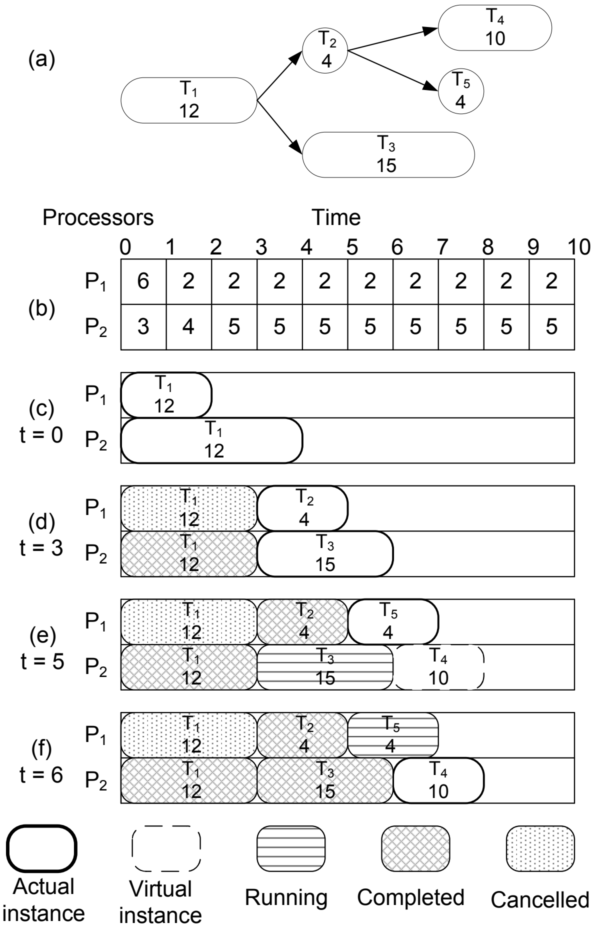
Algorithm 1 Adaptive Scheduling Algorithm

```
compute ranks for all nodes
$\forall k = 1, 2, ..., K$, initialize $C_k$ and $t_k^{EA} \leftarrow 0$
$\forall i = 1, 2, ..., M$, $R_i \leftarrow 0$
while any unfinished node exists do
    $\forall n_i \in S^R$, $R_i^v \leftarrow R_i$
    $\forall n_i \in S^R$, $t_i^{TF} \leftarrow +\infty$
    $\forall k = 1, 2, ..., K$, $t_k^{TA} \leftarrow t_k^{EA}$
    while there exist any idle processor and unfinished node $i$ in $S^R$ with $R_i < R_{max}$ do
        select a ready node $n_i$
        select processor $P_k$
        $t_k^{TA} \leftarrow \max\{t + t_i^{comm}, t_k^{TA}\} + w_i/C_k$
        $R_i^v \leftarrow R_i^v + 1$ /*the assignment is a virtual instance*/
        if $P_k$ is idle and $t_i^{TF} > t_k^{TA}$ then
            $R_i \leftarrow R_i + 1$ /*the assignment is an actual instance*/
        end if
        if $t_i^{TF} > t_k^{TA}$ then
            $t_i^{TF} \leftarrow t_k^{TA}$
        end if
    end while
    edges are scheduled into message queues and transmitted and actual instances are processed
    wait until any processor becomes idle
    kill all the replicas of the task just completed /* all processors that are computing these replicas also become idle */
    $\forall k = 1, 2, ..., K$, update $C_k$ and $t_k^{EA}$
end while
```

Our objective is to minimize the overall processing time, i.e., makespan of all the applications. Minimizing the makespan of parallel applications is the objective of numerous research projects in parallel computing. Here we address this problem with challenges posed by diverse applications and dynamic resources. Finally, we assume that each processor has adequate storage to store and compute any amount of data. Further, when compared to task execution time, the time taken for making scheduling decisions is negligible.

4 The Proposed Algorithm

This section describes the proposed algorithm, which is shown in Algorithm 1. Previous static algorithms often run into the problem of pre-committing the schedule to the processors. A task then can only be executed on the assigned processors even if a favorable rescheduling is possible on another processor. To avoid unfavorable pre-commitment, scheduling decisions should be committed as late as possible, i.e., tasks should be dynamically assigned to idle processors which require task assignments according to the actual execution progress. An idle processor is a processor on which no task from the scheduler is running. The proposed algorithm dynamically works by iteratively assigning nodes to idle processors in multiple rounds. Each round of scheduling is triggered by an event that a processor finishes processing a task.



Processors	0–1	1–2	2–3	3–4	4–5	5–6	6–7	7–8	8–9	9–10
P1	6	2	2	2	2	2	2	2	2	2
P2	3	4	5	5	5	5	5	5	5	5

Fig. 1. Scheduling of ASA for a task graph on 2 processors. (a) Processors' speeds. (b)-(e) Rounds 1-6.

We use terms *actual instance* and *virtual instance* to assist the scheduling of computation and communication. An actual instance refers to an instance of a node that is actually assigned for execution. In contrast, a virtual instance refers to virtual assignments of nodes which are tentatively allocated in the current round of scheduling. In addition, the algorithm applies task duplication to prevent delayed task completion and reduce communication time. Hence, each node may have multiple actual and virtual instances when task duplication is applied. At the beginning of each round of scheduling, the number of virtual instances of a node n_i, denoted as R_i^v, is initialized as the number of actual instances of the node, denoted as R_i (line 5). Actual instances are executed in processors and thus occupy computing resources. To prevent resource wastage caused by excessive task duplication, we limit the maximum number of actual instances for each node that can be produced, denoted as R_{max}, i.e., $R_i \leq R_{max}$.

We define that a node is *ready* if it is unfinished and has no unfinished parents nodes. Hence, a node being computed is also a ready node. Let set S^R contain all ready nodes. Once there exist idle processors and ready nodes satisfying $R_i < R_{max}$, the scheduling process repeats to assign selected tasks to selected processors (lines 8 - 19). Among all nodes in the ready set, the node with the least number of virtual instances are selected for scheduling (line 9). Ties are broken by selecting the node with the highest rank among the nodes with the same least number of tentative instances. The rank of a node is recursively defined as:

$$rank(n_i) = \bar{w_i} + \max_{1 \leq q \leq Q} \{rank(n_{i_q}) + w_{ij}\} \tag{2}$$

where n_i has Q children and n_{i_q} is the q-th child. The rank of an exit node is equal to

$$rank(n_{exit}) = w^{-}_{exit} \tag{3}$$

Basically, $rank(n_i)$ is the longest distance from n_i to an exit node.

Although tasks are only assigned to idle processors, the scheduler still globally considers both idle and busy processors and makes tentative scheduling decisions in order to prevent inefficient scheduling decisions (e.g., assigning key nodes to idle but slow processors). Among all processors, the node is scheduled to the processor P_k with the minimum $\max\{t + t_i^{comm}, t_k^{TA}\} + w_i/C_k$ (line 10). That is, the node is assigned to the processor on which the estimated earliest completion time is achieved.

$$t_k^{TA} = \max\{t + t_i^{comm}, t_k^{TA}\} + w_i/C_k \tag{4}$$

where $\max\{t + t_i^{comm}, t_k^{TA}\}$ is the estimated start time on P_k; t is current time; t_i^{comm} is the estimated time for transferring the edges for n_i, which is obtained as:

$$t_i^{comm} = \max_{1 \leq p \leq P} \{w_{i_p i} r(P(n_{i_p}), P_k)\} \tag{5}$$

where n_i has P parents and n_{i_p} is the p-th parent node of n_i; $P(n_{i_p})$ is the processor which holds n_{i_p}; $r(P(n_{i_p}), P_k) = 1$ if $P(n_{i_p}) \neq P_k$ and zero otherwise. After the node is scheduled on a selected processor P_k, t_k^{TA} is updated accordingly. Also, R_i^v is incremented by one (lines 11, 12).

The scheduler prefers to wait for assigning tasks to busy but fast processors in the future rather than to assign tasks immediately to idle but slow processors if the future assignment can finally cause the task to be finished earlier than the immediate assignment. Such a strategy can effectively prevent unfavorable assignments of mapping critical tasks to slow but idle processors that may finally lead to delayed task completion. Notice that if a processor already has the edges, it may have more opportunities to be selected. Also, the node cannot be scheduled to a processor where a virtual instance of the node already resides.

To avoid inefficient task pre-commitment, only the task assignments to idle processors are immediately issued, i.e., such assignments produce actual instances. Then, the processor is marked as busy and required edges of the task are sent to the processor. The processor will start to process the node once all required edges are prepared. R_i is also incremented by one (lines 13 - 15). On the contrary, the assignments to the busy processors yield virtual instances which are not immediately implemented. A virtual instance may finally be realized in later rounds of scheduling and then becomes an actual instance. That is, if the busy processors' compute capacities are stable and prediction information is accurate, the virtual instance will finally be realized in later rounds when all prior allocated nodes on the processor have been completed. However, if the busy processors are unexpectedly slowed down in the near future, the current tentative assignments will not be realized. Such a prudent strategy can prevent delayed task completion from unexpectedly slowed processors. Because ASA only implement actual replicas, no node is queued in the buffer of each processor. This significantly reduces load unbalancing caused by inaccurate prediction information. Therefore, inaccurate prediction impacts little on the performance of ASA.

We consider an example of scheduling a task graph onto 2 processors, as illustrated in Fig. 1. The task graph with 5 nodes is shown in Fig. 1.(a). Fig. 1.(b) shows the speeds of the processors varying with time. Fig. 1.(c) - (f) illustrates 4 rounds of ASA to complete all tasks at time is 0, 3, 5, and 6, respectively. In this example, communications are assumed to be negligible for the simplicity of illustration. When time is 0, both P_1 and P_2 are idle and the algorithm is executed. The ready task T_1 is first allocated to P_1 since P_1 can finish T_1 earlier than P_2 according to estimation, i.e., P_1 is expected to finish T_1 at time 2 with a speed of 6 while P_2 is expected to finish T_1 at time 4 with a speed of 3. Since T_1 is the only ready task and P_2 is also ready, task duplication is activated, i.e., T_1 is again selected and its replica is assigned to P_2. Both P_1 and P_2 acquire an actual instance of T_1 as they are idle. As the speeds of the processors vary, P_2 first finishes T_1 at time 3 and the execution of T_1 on P_1 is cancelled accordingly. This illustrates the benefits of task duplication in preventing delayed completion. Without the duplication on P_2, P_1 will finish T_1 at time 4, which is much later than the expected time 2. When time is 3, tasks T_2 and T_3 become ready. As T_3 has a higher rank, it is first assigned to P_2. T_2 is then assigned to P_1. When time is 5, P_1 finishes T_2 and becomes idle. At the moment, tasks T_4 and T_5 are ready. T_4 is first selected and tentatively scheduled to the busy processor P_2. This assignment leads to a virtual instance of T_4 as P_2 is busy at this time. Hence, if P_1 unexpectedly speeds and P_2 unexpectedly slows in the future, the virtual instance will not be realized and P_1 still has a chance to be allocated with T_4. Since P_1 is still idle, the next ready task T_5 is then assigned to P_1. Without tentative scheduling, T_4 will be assigned to idle processor P_1 and expected to be finished at time 10. Due to tentative scheduling, the actual finish time is 8. This thus illustrates the advantage of tentative scheduling in preventing unfavorable assignment of key tasks to slow processors.

The algorithm explicitly addresses communication costs and contentions via look-ahead message transmission. This approach transfers prerequisite edges of virtual instances to their destination processors before they become actual instances. Since a virtual instance may be realized as an actual instance in later rounds of scheduling, this look-ahead method can reduce communication waiting time by issuing communications beforehand. In other words, when an actual instance of a node is assigned to a processor, the processor may have received corresponding prerequisite edges and can immediately start to run the node.

However, if the virtual instance is not realized as an actual instance, the prerequisite edges of the instance transferred to destination processors will be wasted. To reduce resource wastage due to excessive look-ahead message transmission, the amount of communications that each processor can transfer for the current round of look-ahead message scheduling is refrained by a limit M_{max}. Let M_k record the amount of communications that will be transferred from a processor P_k for the current round of look-ahead message scheduling. If the scheduler plans to transfer a prerequisite edge e_{ij} of a virtual instance from processor P_k, M_k is increased by w_{ij}. An edge will be placed into the message queue of P_k (where all edges will be actually transmitted) if the edge is required by an actual instance or $M_k < M_{max}$ with the edge counted in. The message queue is a priority queue with two priority levels. An edge required by an actual instance has a high priority otherwise it has a low priority.

As multiple potential source processors may have owned the edge that need be transferred to a destination processor, the scheduler will select one potential source processor to transfer the edge. If the edge is required by a virtual instance, it can only be transferred by a processor P_k satisfying $M_k \leq M_{max}$. Among feasible potential source processors, the one with the fastest link to the destination processor is selected to transfer the edge if information on link speeds is available to the scheduler; otherwise the scheduler randomly selects one source processor to transmit the edge.

Afterwards, the current round terminates. Edges in the message queues are sequentially transferred and actual instances are processed. The scheduler waits until a new processor finishes a task and becomes idle. The scheduler hence updates C_k and t_k^{EA} for all processors and starts a new round. When the scheduler is awaken again, the untransferred prerequisite edges for virtual instances assigned in the last round is cancelled as a new tentative partial schedule will be generated in the new round (lines 20 - 23). The entire execution completes after all tasks are finished.

The worst case complexity of the algorithm is $O(KMR_{max}W_{max})$, where K is the number of processors; M is the number of tasks; R_{max} is the maximum number of replicas per task; W_{max} is the width of the task graph. The complexity is derived as follows: The outer while loop (line 4) will be run at most MR_{max} times since only when a task is finished, the scheduler will be waken up and line 4 will be triggered. The inner while loop (line 8) will be run at most MW_{max} times, which is determined by lines 8-10. W_{max} is the maximum value of S_R.

5 Performance Evaluation

In order to assess the effectiveness of the proposed scheduling policies, we will now present a performance evaluation study, carried out by means of a discrete-event simulator. We compare their performance for a large set of operational scenarios, obtained by combining a set of system configurations with a set of application workloads. The default simulation configurations are set as follows: The number of nodes is 200 and the default graph topology is random. The average size of a node is 20 and the estimated size of a node varies among $[0.5, 1.5]$ times of its actual size. The communication to computation ratio (CCR) is initially set as 0.1. The number of processors is 50. The computation speed of each processor dynamically

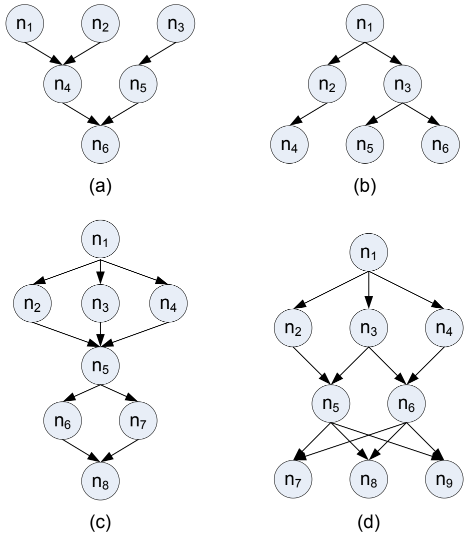


Fig. 2. Miniature example for (a) an in-tree task graph; (b) an out-tree task graph; (c) a fork-join task graph; (d) a work-flow task graph.

varies among $[0.05, speed_{max}]$, where $speed_{max}$ is uniformly distributed in the range $[0.5, 3.5]$. The occurrence of a speed variation event follows an exponential distribution with a mean of $1/\lambda$ where the default value of λ is 0.01. In each of the following experiments we vary one interested parameter while fixing other parameters as their default values to study the effect of the interested parameters. In Figs. 3 and 4, each dot in each figure is the average result of 100 trials.

To understand the merits of ASA we compare it with three baseline algorithms, Choudhury's online algorithm [3], and two offline algorithms, HEFT [10] and ETF [7]. Also, to evaluate the effect of task duplication, we will show the results of ASA with the maximum number of replicas (R) equaling to 0, 1, and 2, respectively. The results will show that making 2 or more replicas is probably cost-ineffective.

The major performance metric is normalized average makespan, defined as the average makespan of an algorithm over that of ASA ($R = 0$). Fig. 3 depicts the normalized average makespan versus λ for five specific task graph topologies: random, in-tree, out-tree, fork-join, and work-flow, respectively, while Fig. 2 shows miniature examples for the specific topologies. In addition, Fig. 4 presents other results on random topologies. Moreover, in Fig. 3(a) error bars are added to denote 95% confidence intervals. The error bars show that the results are rather credible as most results do not deviate.

Figs. 3 shows that ASA ($R = 0, 1, 2$) significantly outperforms the three baseline algorithms by clear margins, which demonstrates the benefits of ASA in handling dynamic task graphs in dynamic environments. Specifically, when λ is quite small, processor speeds rarely vary and thus the offline-estimated processor speeds are comparatively accurate. Accordingly, the performance gaps between ASA and the offline algorithms (HEFT and ETF) are comparatively small. As λ grows, processor speeds frequently vary and thus the offline-estimated processor speeds becomes inaccurate. Therefore, the performance gaps between ASA and the offline algorithms sharply increase since ASA can adapt to dynamically varying processor speeds while the offline algorithms cannot.

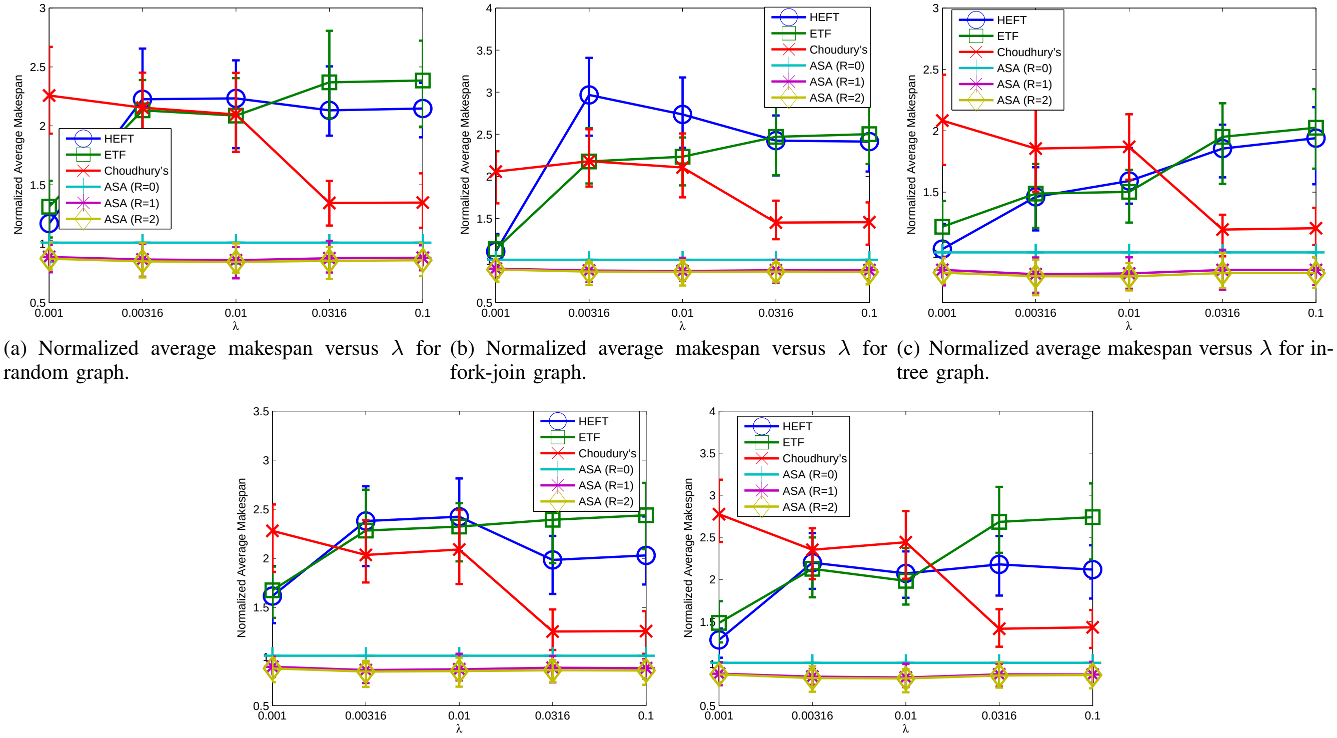
On the other hand, we also observe that the performance gap between ASA and Choudhury's algorithm decreases as λ grows. A plausible explanation is that ASA globally considers all processors in scheduling tasks to avoid assigning critical tasks to idle but slow processors. Accordingly, ASA may prefer to wait for assigning tasks to busy but fast processors in the future rather than immediately assign tasks to idle but slow processors. In contrast, Choudhury's algorithm only considers idle processors in scheduling tasks and thus incurs the risk of assigning key tasks to idle but quite slow processors. This explains why ASA ($R = 0$) outperforms Choudhury's algorithm in the above figures. When λ grows, processor speeds rapidly vary. Current idle but slow processors may become fast soon. Accordingly, the strategy adopted in Choudhury's algorithm that assigning tasks immediately to idle processors may become more efficient.

Furthermore, we can observe that both ASA ($R = 1$) and ASA ($R = 2$) outperform ASA ($R = 0$) for more than 15%, demonstrating the effectiveness of task duplication. Further, ASA ($R = 2$) performs only slightly better than ASA ($R = 1$), showing that making 2 or more replicas is helpless.

In addition, as the proposed algorithm employs duplication techniques, to study possible CPU wastage owing to duplications, Fig. 4(a) plots normalized CPU time usage, which is defined as the average CPU usage of an algorithm divide by that of ASA ($R = 0$). The result shows that ASA ($R = 0$) requires the least CPU time. Since ASA ($R = 1$) and ASA ($R = 2$) require CPU time to process additional replicas, they spend 30% to 45% more CPU time than ASA ($R = 0$). The three baseline algorithms also spend more CPU time than ASA ($R = 0$), because they probably make inefficient scheduling decisions in dynamic computing environments.

Further, we vary CCR from 0.05 to 1.6 and the corresponding normalized makespan is plotted in Fig. 4(b). The result shows that as CCR increases the advantage of ASA over the baseline algorithms decreases. When CCR equals to 1.6, the performance of ASA ($R = 0$) is close to that of the baselines, implying that when communication costs are larger than computation costs, communication-aware strategies are desired in the design of online scheduling algorithms, which can be one direction of the future work.

Moreover, we vary the number of processors from 10 to 160 and the corresponding normalized makespan is plotted in Fig. 4(c). The result shows that as the number of processors grows, the performance gap between ASA and the baselines significantly diminishes. In particular, when the number of processor is 160, Choudhury's algorithm even outperforms all other algorithms. A plausible explanation is that one important benefit of ASA is that it globally considers all processors in scheduling tasks to avoid assigning tasks to idle but quite slow processors. According to the tentative scheduling strategy, ASA may prefer to wait for assigning tasks to busy but fast processors in the future rather than immediately assign tasks to idle but slow processors. When there are abundant processors, probably fast and idle processors exist at any time. In this case, the tentative strategy adopted in ASA becomes



(a) Normalized average makespan versus λ for random graph.

(b) Normalized average makespan versus λ for fork-join graph.

(c) Normalized average makespan versus λ for in-tree graph.

(d) Normalized average makespan versus λ for out-tree graph.

(e) Normalized average makespan versus λ for work-flow graph.

Fig. 3. Normalized average makespan versus various task graph topologies

less competitive. For the same reason, the strategy adopted in Choudhury's algorithm that assigns tasks immediately to idle processors may become more efficient.

To evaluate the scalability of our algorithm, we vary the number of nodes from 50 to 800 and Fig. 4(d) depicts the corresponding normalized makespan. We can observe that as the number of nodes grows, the performance gap between ASA and HEFT/ETF algorithms significantly increases while the performance gap between ASA and Choudhury's online algorithm keeps stable. One possible explanation is that in offline algorithms inefficient scheduling decisions caused by pre-committing can increase the makespan and such increase may tend to accumulate for scheduling large task graphs as large task graphs require more scheduling decisions. In contrast, in online algorithms including ASA and Choudhury's algorithm, such increase cannot easily accumulate since slow processors that finish tasks late will have less chances to receive new tasks. This result demonstrates the benefits of ASA in handling large task graphs.

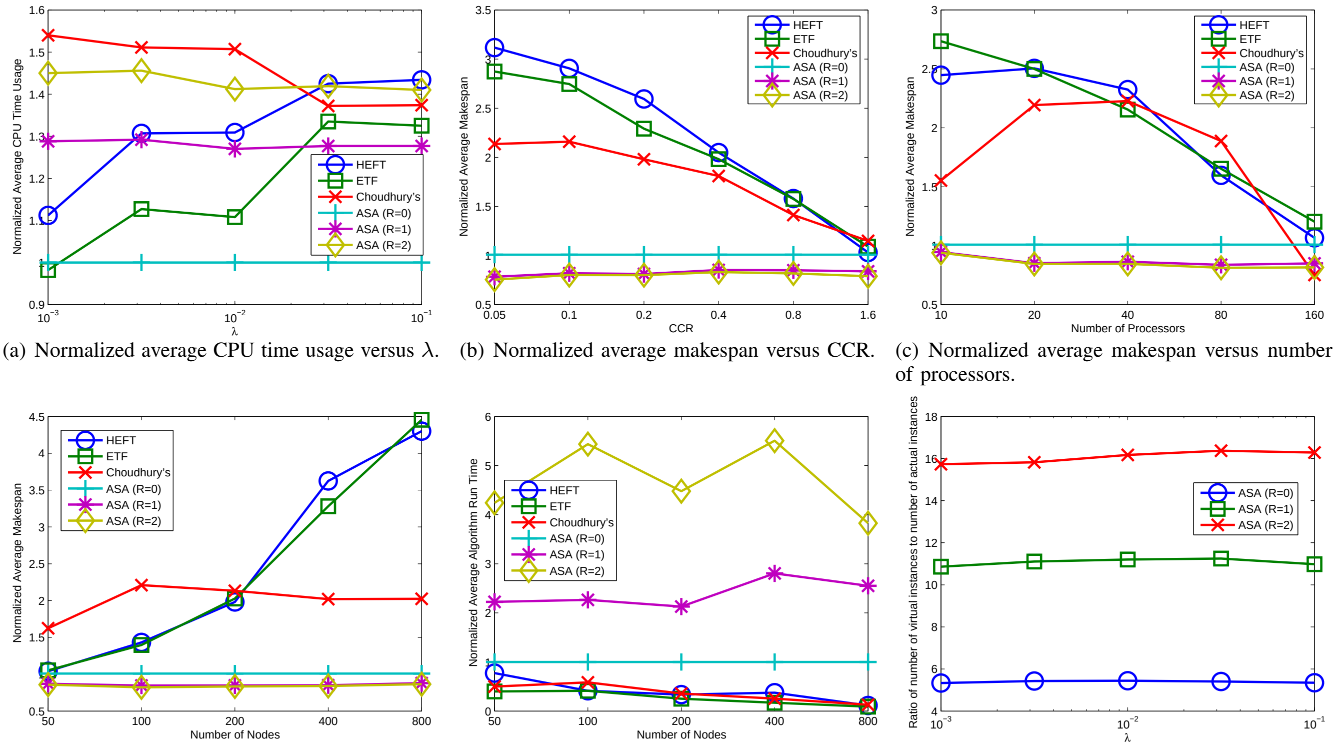
Also, Fig. 4(e) plots normalized algorithm run time with number of nodes varying from 50 to 800. The normalized algorithm run time is defined as the average algorithm run time of an algorithm divide by that of ASA ($R = 0$). Fig. 4(e) shows that ASA spends more time on algorithm execution than the three baselines. When the number of nodes is 50, ASA ($R = 0$) requires up to 2 times of the execution time of the baselines. As the number of nodes increases to 800, ASA ($R = 0$) spends even 10 times of execution time. Also, when R increases from 0 to 2, ASA requires 3 to 4 times of more execution time, showing that making excessive replicas may be cost-ineffective.

Finally, Fig. 4(f) plots ratio of total number of virtual instances to total number of actual instances versus λ. It shows that in ASA, tasks are not assigned to slow processors directly. Instead, the algorithm prefers to assign tasks to fast but busy processors. Thus, a large number of virtual instances (at least 5 times of actual instances) are created for tentative scheduling. In addition, the figure shows that task duplication causes extra virtual instances, which leads to more CPU usage and longer algorithm runtime, as shown in Figs. 4(a) and 4(e).

Lessons learnt from the above results suggests that ASA ($R = 1$) is an efficient choice in the trade-off of performance (in terms of makespan) and costs (in terms of CPU time usage and algorithm run time).

6 Conclusions

In this paper we studied an important problem of scheduling dynamic task graphs on dynamic computing environments. We proposed the ASA algorithm that realistically deals with the dynamic nature of the task graphs and the underlying platforms. The proposed algorithm assigns tasks to idle processors in multiple rounds with a tentative scheduling strategy to prevent inefficient pre-committing decisions and avoid unfavorable assignment of key tasks to slow processors. In addition, the algorithm applies task duplication to prevent delayed task completion. Moreover, a look-ahead message transmission policy is applied to save communication time and thus further improves overall performance. With these useful techniques, the proposed algorithm is resilient to dynamic computing environments and inaccurate profiling information. Our extensive



(a) Normalized average CPU time usage versus λ. (b) Normalized average makespan versus CCR. (c) Normalized average makespan versus number of processors.

(d) Normalized average makespan versus number of nodes. (e) Normalized average algorithm run time versus number of nodes. (f) Ratio of number of virtual instances to number of actual instances versus λ.

Fig. 4. Simulation results on random task graph topologies

simulation results with various test configurations demonstrated the effectiveness and competitiveness of our algorithm. The simulation results show that ASA significantly outperforms three previous peer heuristics under various dynamic scenarios.

References

[1] M.R. Gary and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness. A Series of Books in the Mathematical Sciences.* W.H. Freeman and Co., 1979.

[2] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406-471, 1999.

[3] Pravanjan Choudhury, P.P. Chakrabarti, and Rajeev Kumar, "Online Scheduling of Dynamic Task Graphs with Communication and Contention for Multiprocessors," *IEEE Trans. Parallel Distrib. Systems*, vol. 23, no. 1 pp. 126-133, 2012.

[4] A. Feldmann, M.Y. Kao, J. Sgall, and S.-H. Teng, "Optimal Online Scheduling of Parallel Jobs with Dependencies," *Proc. 25th Ann. ACM Symp. Theory of Computing (STOC 93)* pp. 642-651, 1993.

[5] P. Choudhury, R. Kumar, and P.P. Chakrabarti, "Hybrid Scheduling of Dynamic Task Graphs with Selective Duplication for Multiprocessors under Memory and Time Constraints," *IEEE Trans. Parallel and Distributed Systems*, pp. 967-980, 2008.

[6] N. Fujimoto and K. Hagihara, "Near-Optimal Dynamic Task Scheduling of Independent Coarse-Grained Tasks onto a Computational Grid," *Proc. International Conference on Parallel Processing*, pp. 391-398, 2003.

[7] J.J. Hwang, Y.C. Chow, F.D. Anger and C.Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM Journal of Computing*, vol. 18, no. 2, pp. 244-257, Apr. 1989.

[8] Y. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506-521, May 1996.

[9] M. Wu and D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel and Distributed Systems*, vo. 1, no. 3, pp. 330-343, 1990.

[10] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260-274, MAR. 2002.

[11] V. Kianzad and S.S. Bhattacharyya, "Efficient Techniques for Clustering and Scheduling onto Embedded Multiprocessors," *IEEE Trans. Parallel Distributed System*, vol. 17, no. 7, pp. 667-680, July 2006.

[12] Y.C. Lee and A.Y. Zomaya, "Practical Scheduling of Bag-of-Tasks Applications on Grids with Dynamic Resilience," *IEEE Trans. Computers*, vol. 56, no. 6, pp. 815-825, June 2007.

[13] S. Bansal, P. Kumar, and K. Singh, "Dealing with Heterogeneity through Limited Duplication for Scheduling Precedence Constrained Task Graphs," *J. Parallel and Distributed Computing*, vol. 65, no. 6, pp. 479-491, 2005.

[14] K. Shin, M. Cha, M. Jang, J. Jung, W. Yoon, and S. Choi, "Task Scheduling Algorithm Using Minimized Duplications in Homogeneous Systems," *J. Parallel Distributed Computing*, vol. 68, no. 8, pp. 1146-1156, 2008.

[15] C.I. Park and T.Y. Choe, "An Optimal Scheduling Algorithm Based on Task Duplication," *IEEE Trans. Computers*, vol. 51, no. 4, pp. 444-448, Apr. 2002.

[16] N. Loc and S. Elnaffar, "A Dynamic Scheduling Algorithm for Divisible Loads in Grid Environments," *Journal OF Communications*, vol. 2, no. 4, June 2007.

[17] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, and R. Freund, "Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems," *Proc. Eighth IEEE Heterogeneous Computing Workshop*, pp. 30-44, 1999.

[18] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for Scheduling Parameter Sweep Applications in Grid Environments," *Proc. Ninth Heterogeneous Computing Workshop*, pp. 349-363, May 2000.

[19] E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima, "Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids," Proc. 10-th Workshop Job Scheduling Strategies for Parallel Processing, pp. 210-232, 2004.

[20] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: An Experiment in Public-Resource Computing," *Comm. ACM*, vol. 45, no. 11, pp. 56-61, 2002.