An Efficient Algorithm for Real-Time Divisible Load Scheduling

Anwar Mamat, Ying Lu, Jitender Deogun, Steve Goddard
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE 68588
{anwar, ylu, deogun, goddard}@cse.unl.edu

Abstract

Providing QoS and performance guarantees to arbitrarily divisible loads has become a significant problem for many cluster-based research computing facilities. While progress is being made in scheduling arbitrarily divisible loads, current approaches are not efficient and do not scale well. In this paper, we propose a linear algorithm for real-time divisible load scheduling. Unlike existing approaches, the new algorithm relaxes the tight coupling between the task admission controller and the task dispatcher. By eliminating the need to generate exact schedules in the admission controller, the algorithm avoids high overhead. We experimentally evaluate the new algorithm. Simulation results demonstrate that the algorithm scales well, can schedule large numbers of tasks efficiently, and performs similarly to existing approaches in terms of providing real-time guarantees.

1 Introduction

An arbitrarily divisible or embarrassingly parallel workload can be partitioned into an arbitrarily large number of load fractions. This workload model is a good approximation of many real-world applications [12], e.g., distributed search for a pattern in text, audio, graphical, and database files; distributed processing of big measurement data files; and many simulation problems. All elements in such an application often demand an identical type of processing, and relative to the huge total computation, the processing on each individual element is infinitesimally small. Quite a few scientific applications conform to this divisible load task model. For example, the CMS (Compact Muon Solenoid) [10] and ATLAS (A Toroidal LHC Apparatus) [6] projects, associated with LHC (Large Hadron Collider) at CERN (European Laboratory for Particle Physics), execute cluster-based applications with arbitrarily divisible loads. As such applications become a major type of cluster workloads [27], providing QoS to arbitrarily divisible loads becomes a significant problem for cluster-based research computing facilities like the U.S. CMS Tier-2 sites [28]. By monitoring the CMS mailing-list, we have learned that CMS users always want to know task response times when they submit tasks to clusters. However, without a good QoS mechanism, current cluster sites cannot provide these users good response time estimations.

Table 1: Sizes of OSG Clusters.

<table>
<thead>
<tr>
<th>Host Name</th>
<th>No. of CPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>fermigrid1.fnal.gov</td>
<td>41863</td>
</tr>
<tr>
<td>osgserv01.slac.stanford.edu</td>
<td>9103</td>
</tr>
<tr>
<td>lepton.rcac.purdue.edu</td>
<td>7136</td>
</tr>
<tr>
<td>cmsosgce.fnal.gov</td>
<td>6942</td>
</tr>
<tr>
<td>osggate.clemson.edu</td>
<td>5727</td>
</tr>
<tr>
<td>grid1.oscer.ou.edu</td>
<td>4169</td>
</tr>
<tr>
<td>osg-gw-2.t2.ucsd.edu</td>
<td>3804</td>
</tr>
<tr>
<td>u2-grid.ccr.buffalo.edu</td>
<td>2104</td>
</tr>
<tr>
<td>red.unl.edu</td>
<td>1140</td>
</tr>
</tbody>
</table>

Real-time divisible load scheduling is a well researched area [8, 9, 17, 18, 19, 20]. Focusing on satisfying QoS, providing real-time guarantees, and better utilizing cluster resources, existing approaches give little emphasis to scheduling efficiency. They assume that scheduling takes much less time than the execution of a task, and thus ignore the scheduling overhead.

However, clusters are becoming increasingly bigger and busier. In Table 1, we list the sizes of some OSG (Open Science Grid) clusters. As we can see, all of these clusters have more than one thousand CPUs, with the largest providing over 40 thousand CPUs. Figure 1 shows the number of tasks waiting in the OSG cluster at University of California, San Diego for two 20-hour periods, demonstrating that at times there could be as many as 37 thousand tasks in the waiting queue of a cluster. As the cluster size and workload increase, so does the scheduling overhead. For a cluster with thousands of nodes or thousands of waiting tasks, as will be demonstrated in Section 5, the scheduling overhead could be substantial and existing divisible load scheduling algorithms are no longer applicable due to lack of scalability. For example, to schedule the bursty workload in Figure 1a, the best-known real-time algorithm [8] takes more than 11 hours to make admission control decisions on the 14,000 tasks that arrived in an hour, while our new algorithm needs only 37 minutes.

In this paper, we address the deficiency of existing approaches and present an efficient algorithm for real-time divisible load scheduling. The time complexity of the proposed algorithm is linear in the maximum of the number of tasks in the waiting queue and the number of nodes in the cluster. In addition, the algorithm performs similarly to previous algorithms in terms of providing...
real-time guarantees and utilizing cluster resources.

The remainder of this paper is organized as follows. Related work is presented in Section 2. We describe both task and system models in Section 3. Section 4 discusses the real-time scheduling algorithm and Section 5 evaluates the algorithm performance. We conclude the paper in Section 6.

2 Related Work

Divisible load theory (DLT) has long been studied and applied in distributed systems scheduling [7, 27, 29]. It provides the foundation for optimally partitioning arbitrarily divisible loads to distributed resources. These workloads represent a broad variety of real-world applications in cluster and grid computing, such as BLAST (Basic Local Alignment Search Tool) [2], a bioinformatics application, and high energy and particle physics applications in ATLAS (A Toroidal LHC Apparatus) [6] and CMS (Compact Muon Solenoid) [10] projects. Currently, large clusters usually use batch schedulers [14] to handle their workloads. A Batch scheduler monitors the task execution and sends queued jobs to a cluster node when it becomes available. The goal is to optimize the system utilization. Satisfying QoS and task real-time constraints are, however, not a major objective of such schedulers. Those clusters, therefore, cannot provide users good response time estimations.

The scheduling models investigated for distributed or multiprocessor systems most often (e.g., [1, 5, 15, 16, 23, 25, 26]) assume periodic or aperiodic sequential jobs that must be allocated to a single resource and executed by their deadlines. With the evolution of cluster computing, researchers have begun to investigate real-time scheduling of parallel applications on a cluster [3, 4, 13, 24, 30]. However, most of these studies assume the existence of some form of task graph to describe communication and precedence relations between computational units called subtasks (i.e., nodes in a task graph).

One closely related work is scheduling “scalable real-time tasks” in a multiprocessor system [17]. Similar to a divisible load, a scalable task can be executed on more than one processor and as more processors are allocated to it, its pure computation time decreases. If we use $N$ to represent the number of processors and $n$ to denote the number of tasks waiting in the system, the time complexity of the most efficient algorithms proposed in [17] (i.e., MWF-FA and EDF-FA) is $O(n^2 + nN)$.

There has been some existing work on cluster-based real-time divisible load scheduling [8, 9], including our own previous work [18, 19]. In [19], we developed several scheduling algorithms for real-time divisible loads. Following those algorithms, a task must wait until a sufficient number of processors become available. This could cause a waste of processing power as some processors are idle when the system is waiting for enough processors to become available. This system inefficiency is referred to as the Inserted Idle Times (IITs) [18]. To reduce or completely eliminate IITs, several algorithms have been developed [8, 9, 18], which enable a task to utilize processors at different processor available times. These algorithms lead to better cluster utilizations, but have high scheduling overheads. The time complexity of algorithms proposed in [8, 9] is $O(nN\log N)$ and the algorithm in [18] has a time complexity of $O(nN^3)$.

In this paper, we propose an efficient algorithm for scheduling real-time divisible loads in clusters. Similar to algorithms in [8, 9, 18], our new algorithm eliminates IITs. Furthermore, with a time complexity of $O(\max(N,n))$, the algorithm is efficient and scales well to large clusters.

3 Task and System Models

In this paper, we adopt the same task and system models as those in the previous work [8, 9, 18, 19]. For completeness, we briefly present these below.

**Task Model.** We assume a real-time aperiodic task model in which each aperiodic task $\tau_i$ consists of a single invocation specified by the tuple $(A, \sigma, D)$, where $A$ is the task arrival time, $\sigma$ is the total data size of the task, and $D$ is its relative deadline. The task absolute deadline is given by $A + D$.

**System Model.** A cluster consists of a head node, denoted by $P_0$, connected via a switch to $N$ processing nodes, denoted by $P_1, P_2, \ldots, P_N$. We assume that all processing nodes have the same computational power and all links from the switch to the processing nodes have the same bandwidth. The system model assumes a typical cluster environment in which the head node does not participate in computation. The role of the head node is
to accept or reject incoming tasks, execute the scheduling algorithm, divide the workload and distribute data chunks to processing nodes. Since different nodes process different data chunks, the head node sequentially sends every data chunk to its corresponding processing node via the switch. We assume that data transmission does not occur in parallel. Therefore, only after the head node is calculated by a cost function and deriving the exact schedule, the $\sigma_C$ of processing nodes that of waiting tasks introduces a big of the task may change represents the time to compute a unit of workload $C$ chosen to partition the task among the allocated should be allocated to each task. Third, a strategy is be based on policies like EDF (Earliest Deadline First) or First, it determines the task execution order, which could the algorithm needs to make three important decisions. In this section, we present our new algorithm for schedul-

ing real-time divisible loads in clusters. Due to their spe-

cial property, when scheduling arbitrarily divisible loads, the algorithm needs to make three important decisions. First, it determines the task execution order, which could be based on policies like EDF (Earliest Deadline First) or MWF (Maximum Workload derivative First) [17]. Second, it decides the number $n$ of processing nodes that should be allocated to each task. Third, a strategy is chosen to partition the task among the allocated $n$ nodes.

As is typical for dynamic real-time scheduling algo-

rithms [11, 22, 25], when a task arrives, the scheduler determines if it is feasible to schedule the new task without compromising the guarantees for previously admitted tasks. Only those tasks that pass this schedulability test are allowed to enter the task waiting queue (TWQ). This decision module is referred to as the admission controller. When processing nodes become available, the dispatcher module partitions each task and dispatches subtasks to execute on processing nodes. Both modules, admission controller and dispatcher, run on the head node.

For existing divisible load scheduling algorithms [8, 9, 17, 18, 19], in order to perform the schedulability test, the admission controller generates a new schedule for the newly arrived task and all tasks waiting in TWQ. If the schedule is feasible, the new task is accepted; otherwise, it is rejected. For these algorithms, the dispatcher acts as an execution agent, which simply implements the feasible schedule developed by the admission controller. There are two factors that contribute to large overheads of these algorithms. First, to make an admission control decision, they reschedule tasks in TWQ. Second, they calculate

in the admission controller the minimum number $n_{\min}$ of nodes required to meet a task’s deadline so that it guarantees enough resources for each task. The later a task starts, the more nodes are needed to complete it before its deadline. Therefore, if a task is rescheduled to start at a different time, the $n_{\min}$ of the task may change and needs to be recomputed. This process of rescheduling and recomputing $n_{\min}$ of waiting tasks introduces a big overhead.

To address the deficiency of existing approaches, we develop a new scheduling algorithm, which relaxes the tight coupling between the admission controller and the dispatcher. As a result, the admission controller no longer generates an exact schedule, avoiding the high overhead. To carry out the schedulability test, instead of computing $n_{\min}$ and deriving the exact schedule, the admission controller assumes that tasks are executed one by one with all processing nodes. This simple and efficient all nodes assignment (ANA) policy speeds up the admission control decision. The ANA is, however, impractical. In a real-life cluster, resources are shared and each task is assigned just enough resources to satisfy its needs. For this reason, when dispatching tasks for execution, our dispatcher needs to adopt a different node assignment strategy. If we assume ANA in the admission controller and let the dispatcher apply the minimum node assignment (MNA) policy, we reduce the real-time scheduling overhead but still allow the cluster to have a schedule that is appealing in the practical sense. Furthermore, our dispatcher dispatches a subtask as soon as a processing node and the head node become available, eliminating ITs.

Due to the superior performance of EDF-based divisible load scheduling [19], our new algorithm schedules tasks in EDF order as well. In the following, we describe in detail the two modules of the algorithm: admission controller (Section 4.1) and dispatcher (Section 4.2). Since the two modules follow different rules, sometimes an adjustment of the admission controller is needed to resolve their discrepancy so that task real-time properties can always be guaranteed (Section 4.3). Section 4.4 proves the correctness of our algorithm.

4.1 Admission Controller

When a new task arrives, the admission controller determines if it is feasible to schedule the new task without compromising the guarantees for previously admitted tasks. In the previous work [8, 9, 17, 18, 19, 20], the admission controller follows a brute-force approach, which inserts the new task into TWQ, reschedules each task and generates a new schedule. Depending on the feasibility of the new schedule, the new task is either accepted or rejected. As we can see, both accepting and rejecting a task involve generating a new schedule.

In this paper, we make two significant changes in order to develop a new admission control algorithm. First,

\footnote{Although in this paper, we describe the algorithm assuming EDF scheduling, the idea is applicable to other divisible load scheduling such as MWF-based scheduling algorithms [17].}
to determine the schedulability of a new task, we only check the information recorded with the two adjacent tasks (i.e., the preceding and succeeding tasks). Unlike the previous work, our new algorithm could reject a task without generating a new schedule. This significantly reduces the scheduling overhead for heavily loaded systems. Second, we separate the admission controller from the dispatcher, and to make admission control decisions, an ANA policy is assumed.

The new admission control algorithm is called AC-FAST. Algorithm 1 presents its pseudo code. The admission controller assumes an ANA policy. We use $E$ and $C$ to respectively denote the task execution time and the estimated completion time of task $\tau$. $E(\tau)$ gives the formal definition of $\beta$, which represents the task execution time of running a task $\tau(A, \sigma, D)$ on $N$ processing nodes as [19],

$$E(\tau, N) = \frac{1 - \beta}{1 - \beta^N} \sigma(C_{ms} + C_{ps}),$$  \hspace{1cm} (1)

where $\beta = \frac{C_{ps}}{C_{ms} + C_{ps}}$. \hspace{1cm} (2)

When a new task $\tau$ arrives, the algorithm first checks if the head node $P_0$ will be available early enough to at least finish $\tau$’s data transmission before $\tau$’s absolute deadline. If not so, task $\tau$ is rejected (lines 1-4). As the next step, task $\tau$ is tentatively inserted into TWQ following EDF order and $\tau$’s two adjacent tasks $\tau_s$ and $\tau_p$ (i.e., the succeeding and the preceding tasks) are identified (lines 5-6). By using the information recorded with $\tau_s$ and $\tau_p$, the algorithm further tests the schedulability. First, to check whether accepting $\tau$ will violate the deadline of any admitted task, the algorithm compares $\tau$’s execution time $\tau.E$ with its successor $\tau_s$’s $\text{slack}_{min}$, which represents the minimum slack of all tasks scheduled after $\tau$. Next, we give the formal definition of $\text{slack}_{min}$. Let $S$ denote the task start time. A task’s slack is defined as,

$$\text{slack} = A + D - (S + E),$$ \hspace{1cm} (3)

which reflects the scheduling flexibility of a task. Starting a task slack time units later does not violate its deadline. Therefore, as long as $\tau$’s available execution time is no more than slack of any succeeding task, accepting $\tau$ will not violate any admitted task’s deadline. We define $\tau_i.\text{slack}_{min}$ as the minimum slack of all tasks scheduled after $\tau_{i-1}$. That is,

$$\tau_i.\text{slack}_{min} = \min(\tau_i.\text{slack}, \tau_{i+1}.\text{slack}, \cdots, \tau_n.\text{slack}).$$ \hspace{1cm} (4)

If $\tau$’s execution time is less than its successor $\tau_s$’s $\text{slack}_{min}$, accepting $\tau$ will not violate any task’s deadline (lines 7-10).

The algorithm then checks if task $\tau$’s deadline can be satisfied or not, i.e., to check if $\tau(A + D - S) \geq \tau.E$, where the task start time $\tau.S$ is the preceding task’s completion time $\tau_p.C$ or $\tau$’s arrival time $\tau.A$ (lines 11-31). If there is a task in TWQ, then the cluster is busy. For a busy cluster, we do not need to resolve the discrepancy between the admission controller and the dispatcher and the task real-time properties are still guaranteed (see Section 4.4 for a proof). However, if TWQ becomes empty, the available resources could become idle and the admission controller must consider this resource idleness. As a result, in our AC-FAST algorithm, when a new task $\tau$ arrives into an empty TWQ, an adjustment is made (lines 15-17). The purpose is to resolve the discrepancy between the admission controller and the dispatcher so that the number of tasks admitted will not exceed the cluster capacity. For a detailed discussion of this adjustment, please refer to Section 4.3. Once a new task $\tau$ is admitted, the algorithm inserts $\tau$ into TWQ and modifies the $\text{slack}_{min}$ and the estimated completion time of tasks scheduled after $\tau$ (lines 22-31).

**Time Complexity Analysis.** In our AC-FAST algorithm, the schedulability test is done by checking the information recorded with the two adjacent tasks. Since TWQ is sorted, locating $\tau$’s insertion point takes $O(\log(n))$ time and so do functions $\text{getPredecessor}(\tau)$ and $\text{getSuccessor}(\tau)$. Function adjust($\tau$) runs in $O(N)$ time (see Section 4.3) and it only occurs when TWQ is empty. The time complexity of function updateSlacks is $O(n)$. Therefore, algorithm AC-FAST has a linear i.e., $O(\max(N, n))$ time complexity.

### 4.2 Dispatcher

The dispatching algorithm is rather straightforward. When a processing node and the head node become available, the dispatcher takes the first task $\tau(A, \sigma, D)$ in TWQ, partitions the task and sends a subtask of size $\hat{\sigma}$ to the node, where $\hat{\sigma} = \min\left(\frac{D - \text{CurrentTime}}{C_{ms} + C_{ps}}, \sigma\right)$. The remaining portion of the task $\tau(A, \sigma - \hat{\sigma}, D)$ is left in TWQ. As we can see, the dispatcher chooses a proper size $\hat{\sigma}$ to guarantee that the dispatched subtask completes no later than the task’s absolute deadline $A + D$. Following the algorithm, all subtasks of a given task complete at the task absolute deadline, except for the last one, which may not be big enough to occupy the node until the task deadline. By dispatching the task as soon as the resources become available and letting the task occupy the node until the task deadline, the dispatcher allocates the minimum number of nodes to each task.

To illustrate by an example, if two tasks $\tau_1$ and $\tau_2$ are put into TWQ, from the admission controller’s point of view, they will execute one by one using all nodes of the cluster (see Figure 2a); in reality, they are dispatched and executed as shown in Figure 2b, occupying the minimum numbers of nodes needed to meet their deadline requirements.

### 4.3 Admission Controller Adjustment

As discussed in previous sections, the admission controller assumes a different schedule than the one adopted by the dispatcher. If TWQ is not empty, the resources are
Algorithm 1 AC-FAST(τ(A, σ, D), TWQ)
1: //check head node’s available time
2: if (τ(A + D) ≤ P0.AvailableTime + τ.σCmin) then
3:    return false
4: end if
5: τp = getPredecessor(τ)
6: τs = getSuccessor(τ)
7: τ.E = E(τ, σ, N)
8: if (τs ≠ null && τ.E > τ.s.slackmin) then
9:    return false
10: end if
11: if (τp == null) then
12:    τ.S = τ.A
13: else
14:    τ.S = τp.C
15: end if
16: if (TWQ == ∅) then
17:    adjust(τ)
18: end if
19: τ.S = max(τ.S, τ.A)
20: if τ.(A + D − S) < τ.E then
21:    return false
22: else
23:    τ.slack = τ.(A + D − S − E)
24:    τ.C = τ.(S + E)
25: end if
26: TWQ.insert(τ)
27: updateSlacks(τ, TWQ)
28: for (τi ∈ TWQ && τi.(A + D) > τ.(A + D)) do
29:    τi.C+ = τ.E
30: end for
31: return true
32: end if

Algorithm 2 updateSlacks(τ(A, σ, D), TWQ)
1: for (τi ∈ TWQ ) do
2:    if (τi.(A + D) > τ.(A + D)) then
3:        τi.slack = τi.slack − τ.E
4:    end if
5: end for
6: i = TWQ.length;
7: τi.slackmin = τi.slack
8: for (i = TWQ.length - 1; i ≥ 1; i −−) do
9:    τi.slackmin = min(τi.slack, τi+1.slackmin)
10: end for

always utilized. In this case, the admission controller can make correct decisions assuming the ANA policy without detailed knowledge of the system. The admitted tasks are dispatched following the MNA policy and are always successfully completed by their deadlines. However, if TWQ is empty, some resources may be idle until the next task arrival. At that point, the admission controller has to know the system status so that it takes resource idleness into account to make correct admission control decisions.

We illustrate this problem in Figure 3. τ1 arrives at time 0. The admission controller accepts it and estimates it to complete at time 7 (Figure 3a). However, because τ1 has a loose deadline, the dispatcher does not allocate all four nodes but the minimum number, one node, to τ1 and completes it at time 20 (Figure 3b). Task τ2 arrives at an empty TWQ at time 6 with an absolute deadline of 14. The nodes P2, P3, P4 are idle during the time interval [4, 6]. If the admission controller were not to consider this resource idleness, it would assume that all four nodes are busy processing τ1 during the interval [4, 6] and are available during the interval [7, 14]. And thus, it would wrongly conclude that τ2 can be finished with all four nodes before its deadline. However, if τ2 were accepted, the dispatcher cannot allocate all four nodes to τ2 at time 6, because node P1 is still busy processing τ1. With just three nodes available during the interval [6, 20], τ2 cannot complete until time 15 and misses its deadline.

To solve this problem, when a new task arrives at an empty TWQ, the admission controller invokes Algorithm 3 to compute the idle time and make a proper adjustment. The algorithm first computes the workload (σidle) that could have been processed using the idled resources (lines 1-6). According to Eq (1), we know, with all N nodes, it takes w = 1−β−[σidle](Cmax + Cps) time units to execute the workload σidle (line 7). To consider this idle time effect, the admission controller inserts an
Algorithm 3 adjust(\(\tau\))

1: TotalIdle = 0
2: for \((i = 1; i \leq N; i++\)) do
3: \(\tau = \max(P_i.\text{AvailableTime}, P_h.\text{AvailableTime})\)
4: TotalIdle += max\((A - \tau, 0)\)
5: end for
6: \(\sigma_{idle} = \frac{\text{TotalIdle}}{C_{ms} + C_{ps}}\)
7: \(w = \frac{1 - \beta}{2} \sigma_{idle}(C_{ms} + C_{ps})\)
8: \(\tau, S + = w\)

idle task of size \(\sigma_{idle}\) before \(\tau\) and postpones \(\tau\)'s start time by \(w\) (line 8).

4.4 Correctness of the Algorithm

In this section, we prove all tasks that have been admitted by the admission controller can be dispatched successfully by the dispatcher and finished before their deadlines. For simplicity, in this section, we use \(A_i, \sigma_i,\) and \(D_i\) to respectively denote the arrival time, the data size, and the relative deadline of task \(\tau_i\). We prove by contradiction that no admitted task misses its deadline. Let us assume \(\tau_m\) is the first task in TWQ that misses its deadline at \(d_m = A_m + D_m\). We also assume that tasks \(\tau_0, \tau_1, \ldots, \tau_{m-1}\) have been executed before \(\tau_m\). Among these preceding tasks, let \(\hat{\tau}_b\) be the latest that has arrived at an empty cluster. That is, tasks \(\tau_{b+1}, \tau_{b+2}, \ldots, \tau_m\) have all arrived at times when there is at least one task executing in the cluster. Since only tasks that are assumed to finish by their deadlines are admitted, tasks execute in EDF order, and \(\hat{\tau}_b, \tau_{b+1}, \ldots, \tau_m\) are all admitted tasks, we know that the admission controller has assumed that all these tasks can complete by \(\tau_m\)'s deadline \(d_m\). Let \(\sigma^AN\) denote the total workload that has been admitted to execute in the time interval \([A_b, d_m]\). We have,

\[
\sigma^AN = \sum_{i=1}^{m} \sigma_i. \tag{5}
\]

Since all dispatched subtasks are guaranteed to finish by their deadlines (Section 4.2), task \(\tau_m\) missing its deadline means at time \(d_m\), a portion of \(\tau_m\) is still in TWQ. That is, the total workload \(\sigma^MN\) dispatched in the time interval \([A_b, d_m]\) must be less than \(\sum_{i=1}^{m} \sigma_i\). With Eq (5), we have,

\[
\sigma^AN > \sigma^MN. \tag{6}
\]

Next, we prove that Eq (6) cannot hold.

As mentioned earlier, tasks \(\tau_{b+1}, \tau_{b+2}, \ldots, \tau_m\) have all arrived at times when there is at least one task executing in the cluster. However, at their arrival times, TWQ could be empty. As described in Section 4.3, when a task arrives at an empty TWQ, an adjustment function is invoked to allow the admission controller to take resource idleness into account. Following the function (Algorithm 3), the admission controller properly postpones the new task \(\tau\)'s start time by \(w\), which is equivalent to the case where the admission controller “admits” and inserts before \(\tau\) an idle task \(\tau_{idle}\) of size \(\sigma_{idle}\) that completely “occupies” the idled resources present in the cluster. Let us assume that \(\tau_1, \tau_2, \ldots, \tau_i\) are the idle tasks “admitted” by the admission controller adjustment function to “complete” in the interval \([A_b, d_m]\).

We define \(\hat{\sigma}^AN\) as the total workload, including those \(\hat{\sigma}_i, i = 1, 2, \ldots, v\) of idle tasks, that has been admitted to execute in the time interval \([A_b, d_m]\). \(\hat{\sigma}^MN\) is the total workload, including those \(\hat{\sigma}_i, i = 1, 2, \ldots, v\) of idle tasks, that has been dispatched in the time interval \([A_b, d_m]\). Then, we have,

\[
\hat{\sigma}^AN = \sigma^AN + \sum_{i=1}^{v} \hat{\sigma}_i, \tag{7}
\]

\[
\hat{\sigma}^MN = \sigma^MN + \sum_{i=1}^{v} \hat{\sigma}_i. \tag{8}
\]

Next, we first prove that \(\hat{\sigma}^MN \geq \hat{\sigma}^AN\) is true. Due to the space limitation, in this paper, we only provide the sketch of the proof. For detailed derivation and proof, please refer to our technical report [21].

Computation of \(\hat{\sigma}^AN, \hat{\sigma}^AN\) is the sum of workloads, including those \(\sum_{i=1}^{v} \sigma_i\) of idle tasks, that are admitted to execute in the time interval \([A_b, d_m]\). To compute \(\hat{\sigma}^AN\), we leverage the following lemma from [21].

Lemma 4.1 [21] For an admission controller that assumes the ANA policy, if \(h\) admitted tasks are merged into one task \(T\), task \(T\)'s execution time is equal to the sum of all \(h\) tasks’ execution times. That is,

\[
\mathcal{E}(\sum_{i=1}^{h} \sigma_i, N) = \sum_{i=1}^{h} \mathcal{E}(\sigma_i, N). \tag{9}
\]

Figure 4: Merging Multiple Tasks into One Task.

Since \(\hat{\sigma}^AN = \sigma^AN + \sum_{i=1}^{v} \hat{\sigma}_i\), according to the lemma, we have \(\mathcal{E}(\hat{\sigma}^AN, N) = \mathcal{E}(\sigma^AN, N) + \sum_{i=1}^{v} \mathcal{E}(\hat{\sigma}_i, N)\), which implies that the sum of workloads \(\hat{\sigma}^AN\) admitted to execute in the interval \([A_b, d_m]\), equals to the size of the single workload that can be processed by the \(N\) nodes in \([A_b, d_m]\). According to Eq (1), we have

\[
\hat{\sigma}^AN = \frac{d_m - A_b}{1 - \beta} (C_{ps} + C_{ms}). \tag{10}
\]

In addition, it is the sum of workloads assumed to be assigned to each of the \(N\) nodes in the interval \([A_b, d_m]\). We use \(\sigma_{ps}\) to denote the workload fraction assumed to
be processed by node \( P_k \) in the interval \([A_k, d_m]\). Thus, as shown in Figure 5, we have,

\[
\hat{\sigma}^{AN} = \sum_{k=1}^{N} \sigma_{p_k}.
\] (11)

\[
P_k\left/\begin{array}{c}
\sigma_{C_m} \\
\sigma_{p} \\
\sigma_{C_m}
\end{array}\right/ \ 
\begin{array}{c}
P_m \\
B_m
\end{array}
\]

**Figure 5: All Node Assignment Scenario.**

**Computation of \( \hat{\sigma}^{MN} \);** \( \hat{\sigma}^{MN} \) denotes the total workload processed in the time interval \([A_b, d_m]\). With idle tasks \( \tau_1, \tau_2, \cdots, \tau_\ell \) completely “occupying” the idle resources during the interval \([A_b, d_m]\), there are no gaps between “task executions” and the cluster is always “busy” processing \( \hat{\sigma}^{MN} = \sigma_{\sigma}^{MN} + \sum_{i=1}^{\ell} \sigma_{\ell} \). Similar to computing \( \hat{\sigma}^{AN} \), we calculate how much workloads are processed by each of the \( N \) nodes in the given interval. We use \( \sigma_{p_k} \) to denote the sum of workloads that are processed by node \( P_k \) in the interval \([A_k, d_m]\). We have,

\[
\hat{\sigma}^{MN} = \sum_{k=1}^{N} \sigma_{p_k}'.
\] (12)

As shown in [21], \( \sum_{k=1}^{N} \sigma_{p_k}' \geq \sum_{k=1}^{N} \sigma_{p_k} \). Thus, we have,

\[
\hat{\sigma}^{MN} \geq \hat{\sigma}^{AN}.
\] (13)

With Equations (13), (7), and (8), we conclude that \( \sigma^{MN} \geq \sigma^{AN} \) is true, which contradicts Eq (6). Therefore, the original assumption does not hold and no task misses its deadline.

### 5 Evaluation

In the previous section, we presented an efficient divisible load scheduling algorithm. Since the algorithm is based on EDF scheduling and it eliminates IITs, we use FAST-EDF-IIT to denote it. The EDF-based algorithm proposed in [18] is represented by EDF-IIT-1 and that in [8] by EDF-IIT-2. This section evaluates their performance.

We have developed a discrete simulator, called DLSim, to simulate real-time divisible load scheduling in clusters. This simulator, implemented in Java, is a component-based tool, where the main components include a workload generator, a cluster configuration component, a real-time scheduler, and a logging component. For every simulation, three parameters, \( N, C_{ms} \), and \( C_{ps} \) are specified for a cluster.

#### 5.1 Real-Time Performance

We first evaluate the algorithm’s real-time performance. The workload is generated following the same approach as described in [18, 19] and due to the space limitation, we choose not to repeat the details here. Similar to the work by Lee et al. [17], we adopt a metric SystemLoad = \( \mathcal{E}(Avg_e, 1) \) to represent how loaded a cluster is for a simulation, where \( Avg_e \) is the average task data size, \( \mathcal{E}(Avg_e, 1) \) is the execution time of running an average size task on a single node (see Eq (1) for \( \mathcal{E} \)'s calculation), and \( \frac{1}{\lambda} \) is the average task arrival rate per node. To evaluate the real-time performance, we use two metrics—Task Reject Ratio and System Utilization. Task reject ratio is the ratio of the number of task rejections to the number of task arrivals. The smaller the ratio, the better the performance. In contrast, the greater the system utilization, the better the performance.

For the simulations in this subsection, we assume that the cluster is lightly loaded and thus we can ignore the scheduling overheads. In these simulations, we observe that all admitted tasks complete successfully by their deadlines. Figure 6 illustrates the algorithm’s Task Reject Ratio and System Utilization. As we can see, among the three algorithms, EDF-IIT-2 provides the best real-time performance, achieving the least Task Reject Ratio and the highest System Utilization, while FAST-EDF-IIT performs better than EDF-IIT-1. The reason that FAST-EDF-IIT does not have the best real-time performance is due to its admission controller’s slightly pessimistic estimates of the data transmission blocking time (Section 4). Focusing on reducing the scheduling overhead, FAST-EDF-IIT trades real-time performance for algorithm efficiency. In the next subsection, we use experimental data to demonstrate that in busy clusters with long task waiting queues, scheduling overheads become significant and inefficient algorithms like EDF-IIT-1 and EDF-IIT-2 can no longer be applied, while FAST-EDF-IIT wins for its huge advantages in scheduling efficiency.

#### 5.2 Scheduling Overhead

A second group of simulations are carried out to evaluate the overhead of the scheduling algorithms. Before discussing the simulations, we first present some typical cluster workloads, which lay out the rationale for our simulations.

In Figure 1, we have shown the TWQ status of a cluster at University of California, San Diego. From the curves, we observe that 1) waiting tasks could increase from 3,000 to 17,000 in one hour (Figure 1a) and increase from 15,000 to 25,000 in about three hours (Figure 1b) and 2) during busy hours, there could be on average more than 5,000 and a maximum of 37,000 tasks waiting in a cluster. Similarly busy and bursty workloads have also been observed in other clusters (Figure 7) and are quite common phenomena.\(^3\) Based on these typical workload patterns, we design our simulations and evaluate the algorithm’s scheduling overhead.

\(^3\)To illustrate the intensity and commonness of the phenomena, Figures 1 and 7 show the TWQ statistics on an hourly and a daily basis respectively.
In this group of simulations, the following parameters are set for the cluster: N=512 or 1024, \( C_{\text{max}} = 1 \) and \( C_p = 1000 \). We choose to simulate modest-size clusters (i.e., those with 512 or 1024 nodes). According to our analysis, the time complexities of algorithms FAST-EDF-IIT, EDF-IIT-1 and EDF-IIT-2 are respectively \( O(\max(N, n)) \), \( O(nN^3) \) and \( O(nN\log(N)) \). Therefore, if we show by simulation data that in modest-size clusters of N=512 or 1024 nodes FAST-EDF-IIT leads to much less overheads, then we know for sure that it will be even more advantageous if we apply it in larger clusters like those listed in Table 1.

To create cases where we have a large number of tasks in TWQ, we first submit a huge task to the cluster. Since it takes the cluster a long time to finish processing this one task, we can submit thousands of other tasks and get them queued up in TWQ. As new tasks arrive, the TWQ length is built up. In order to control the number of waiting tasks and create the same TWQ lengths for the three scheduling algorithms, tasks are assigned long deadlines so that they will all be admitted and put into TWQ. That is, in this group of simulations, we force task reject ratios to be 0 for all three algorithms so that the measured scheduling overheads of the three are comparable.

We first measure the average scheduling time of the first \( n \) tasks, where \( n \) is in the range \([100, 3000]\). The simulation results for the 512-node cluster are shown in Table 2 and Figure 8. From the data, we can see that for the first 3,000 tasks, FAST-EDF-IIT spends an average of 48.87ms to admit a task, while EDF-IIT-1 and EDF-IIT-2 average respectively 6206.91ms and 1494.91ms, 127 and 30 times longer than FAST-EDF-IIT.

Because the scheduling overhead increases with the number of tasks in TWQ, we then measure the task scheduling time after \( n \) tasks are queued up in TWQ. Table 3 shows the average scheduling time of 100 new tasks after there are already \( n \) tasks in TWQ of the 512-node cluster. The corresponding curves are in Figure 9. As
shown, when there are 3,000 waiting tasks, FAST-EDF-IIT takes 157 ms to admit a task, while EDF-IIT-1 and EDF-IIT-2 respectively spend about 31 and 3 seconds to make an admission control decision.

Table 3: 512-Node Cluster: Average Task Scheduling Time (ms) after \( n \) Tasks in TWQ.

<table>
<thead>
<tr>
<th>( n )</th>
<th>FAST-EDF-IIT</th>
<th>EDF-IIT-1</th>
<th>EDF-IIT-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>1.71</td>
<td>850.01</td>
<td>349.22</td>
</tr>
<tr>
<td>1000</td>
<td>16.25</td>
<td>3006.01</td>
<td>1034.21</td>
</tr>
<tr>
<td>2000</td>
<td>67.24</td>
<td>7536.32</td>
<td>2030.48</td>
</tr>
<tr>
<td>3000</td>
<td>157</td>
<td>31173.86</td>
<td>3089.6</td>
</tr>
</tbody>
</table>

Figure 9: 512-Node Cluster: Algorithm’s Real-Time Scheduling Overhead: Average Scheduling Time after \( n \) Tasks in TWQ.

Now, let us examine the simulation results and analyze their implication for real-world clusters. It is shown in Figure 1a that the length of TWQ in a cluster could increase from 3,000 to 17,000 in an hour. From Table 3, we know that for EDF-IIT-1 and EDF-IIT-2, it takes respectively more than 31 and 3 seconds to admit a task when the TWQ length is over 3,000. Therefore, to schedule the 14,000 new tasks arrived in that hour, it takes more than 7,000 and 700 minutes respectively. Even if we assume that the last one of the 14,000 tasks has arrived in the last minute of the hour, its user has to wait for at least 700-60=640 minutes to know if the task is admitted or not. On the other hand, if FAST-EDF-IIT is applied, it takes a total of 37 minutes to make admission control decisions on the 14,000 tasks. This example demonstrates that our new algorithm is much more efficient than existing approaches and is the only algorithm that can be applied in busy clusters. If we analyze the algorithms using data in Figure 1b where waiting tasks increase from 15,000 to 25,000, the difference in scheduling time will be even more striking.

The simulation results for the 1024-node cluster are reported in Table 4 and Figure 10. Due to EDF-IIT-1’s huge overhead and cubic complexity with respect to the number of nodes in the cluster, a simulation for a busy cluster with a thousand nodes would take weeks — with no new knowledge to be learned from the experiment. Therefore, on the 1024-node cluster, we only simulate EDF-IIT-2 and FAST-EDF-IIT. For easy comparison,

Table 4: First \( n \) Tasks’ Average Scheduling Time (ms).

<table>
<thead>
<tr>
<th>( n )</th>
<th>FAST-EDF-IIT</th>
<th>EDF-IIT-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N=1024 )</td>
<td>( N=512 )</td>
<td>( N=1024 )</td>
</tr>
<tr>
<td>300</td>
<td>1.01</td>
<td>0.96</td>
</tr>
<tr>
<td>1000</td>
<td>4.90</td>
<td>4.84</td>
</tr>
<tr>
<td>2000</td>
<td>21.1</td>
<td>20.46</td>
</tr>
<tr>
<td>3000</td>
<td>50</td>
<td>48.87</td>
</tr>
</tbody>
</table>

Figure 10: Algorithm’s Real-Time Scheduling Overhead: First \( n \) Tasks’ Average Scheduling Time.

Table 4 and Figure 10 include not only data for the 1024-node cluster but also those for the 512-node cluster. As shown by the simulation results, when the cluster size increases from 512 to 1024 nodes, the scheduling overhead of FAST-EDF-IIT only increases slightly. FAST-EDF-IIT has a time complexity of \( O(\max(N,n)) \). Therefore, for busy clusters with thousands of tasks in TWQ (i.e., \( n \) in the range \([3000, 17000]\)), the cluster size increase does not lead to a big increase of FAST-EDF-IIT’s overhead. In contrast, EDF-IIT-2, with a time complexity of \( O(nN\text{log}N) \), has a much larger scheduling overhead on the 1024-node cluster than that on the 512-node cluster.

6 Conclusion

This paper presents a novel algorithm for scheduling real-time divisible loads in clusters. The algorithm assumes a different scheduling rule in the admission controller than that adopted by the dispatcher. Since the admission controller no longer generates an exact schedule, the scheduling overhead is reduced significantly. Unlike the previous approaches, where time complexities are \( O(nN^3) \) [18] and \( O(nN\text{log}N) \) [8], our new algorithm has a time complexity of \( O(\max(N,n)) \). We prove that the proposed algorithm is correct, provides admitted tasks real-time guarantees, and utilizes cluster resources well. We experimentally compare our algorithm with existing approaches. Simulation results demonstrate that it scales well and can schedule large numbers of tasks efficiently. With growing cluster sizes, we expect our algorithm to be even more advantageous.
References


