Efficient Real-Time Divisible Load Scheduling with Advance Reservations

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. With the emergence of Grid applications that require simultaneous access to multi-site resources, supporting advance reservations in a cluster has become increasingly important. While progress is being made in scheduling arbitrarily divisible loads, current approaches either do not support advance reservations, or are not efficient. In this paper, we propose a linear algorithm for real-time divisible load scheduling that supports advance reservations in a cluster. 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. Our approach also addresses the under-utilization concerns raised by advance reservations.

I. INTRODUCTION

Arbitrarily divisible or embarrassingly parallel workloads can be partitioned into an arbitrarily large number of independent load fractions, and are quite common in bioinformatics as well as high energy and particle physics. For example, the CMS (Compact Muon Solenoid) [1] and ATLAS (A Toroidal LHC Apparatus) [2] projects, associated with the Large Hadron Collider (LHC) at CERN (European Laboratory for Particle Physics), execute cluster-based applications with arbitrarily divisible loads.

In a large-scale cluster, the resource management system (RMS), which provides real-time guarantees or QoS, is central to its operation. As a result, the real-time scheduling of arbitrarily divisible loads is becoming a significant problem for cluster-based research computing facilities like the U.S. CMS Tier-2 sites [3]. Due to the increasing importance [4], a few efforts [5], [6], [7] have been made in real-time divisible load scheduling, with significant initial progress in important theories and novel approaches.

To support real-time applications at a Grid level, advance reservations of cluster resources play a key role. Cluster-based Real-time divisible load scheduling is investigated in [8], [9], [6], [10], [7]. However, in a cluster, advance reservations have been largely ignored due to the under-utilization concerns and lack of support for agreement enforcement [11]. In a cluster with no reservation, resources are allocated to tasks until they finish processing. If, however, advance reservations are supported in a cluster, computing nodes and the communication channel could be reserved for a period of time and become unavailable for regular tasks. Due to these constraints, it becomes a very difficult task to count the available resources and schedule real-time tasks. Real-time divisible load scheduling with advance reservations was first investigated in [12], where a multi-stage algorithm was proposed.

Focusing on satisfying QoS, providing real-time guarantees, and better utilizing cluster resources, the existing approach gives little emphasis to scheduling efficiency. It assumes that scheduling takes much less time than the execution of a task, and thus ignores the scheduling overhead. However, clusters are becoming increasingly bigger and busier. Most OSG (Open Science Grid) clusters have more than one thousand CPUs, with the largest providing over 40 thousand CPUs [13]. 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, the scheduling overhead could be substantial and the existing divisible load scheduling algorithm in [12] is no longer applicable due to the lack of scalability.

In this paper, we address the deficiency of the existing approach in [12] and present an efficient algorithm for real-time divisible load scheduling that supports advance
reservations. 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.

II. TASK AND SYSTEM MODELS

In this paper, we adopt similar task and system models as our previous work [7], [12]. For completeness, we briefly present these below.

**Task Model.** There are two types of task: the reservation and the regular task. A reservation $R_i$ is specified by the tuple $(R_{a_i}, R_{e_i}, n_i, R_{i}^{e}, IO_{ratio}^{i})$, where $R_{a_i}$ is the arrival time of the reservation request, $R_{e_i}$ is the start time and the finish time of the reservation, $n_i$ is the number of nodes to be reserved in $[R_{a_i}, R_{e_i}]$ interval, and $IO_{ratio}^{i}$ specifies the data transmission time relative to the length of reservations. It is assumed that for a reservation, data transmission happens at the beginning and computation follows. Let $R_{io}^{i} = R_{e_i} - R_{a_i} + R_{io}^{e_i} - R_{io}^{e_i} \times IO_{ratio}^{i}$. We have data transmission in the interval $[R_{io}^{a_i}, R_{io}^{e_i}]$ and computation in the interval $[R_{io}^{a_i}, R_{io}^{e_i}]$. For a regular (non-reservation) task, a real-time aperiodic task model is assumed, in which each aperiodic task $T_i$ consists of a single invocation specified by $(A_i, \sigma_i, D_i)$, where $A_i$ is the task arrival time, $\sigma_i$ is the total data size of the task, and $D_i$ is its relative deadline. The task absolute deadline is given by $A_i + D_i$. Assuming $T_i$ is arbitrarily divisible, the task execution time is thus dynamically computed based on total data size $\sigma_i$, resources allocated (i.e., processing nodes and bandwidth) and the partitioning method applied to parallelize the computation.

**System Model.** A cluster consists of a head node, denoted by $P_0$, connected via a switch to $N$ processing nodes, $P_1, P_2, \ldots, P_N$. We assume that all processing nodes have the same computational power and bandwidth to the switch. 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. For the arbitrarily divisible loads, tasks and subtasks are independent. Therefore, there is no need for processing nodes to communicate with each other.

According to divisible load theory [14], linear models are used to represent processing and transmission times of regular tasks. In the simplest scenario, the computation time of a load $\sigma$ is calculated by a cost function $Cp(\sigma) = \sigma C_{ps}$, where $C_{ps}$ represents the time to compute a unit of workload on a single processing node. The transmission time of a load $\sigma$ is calculated by a cost function $Cm(\sigma) = \sigma C_{ms}$, where $C_{ms}$ is the time to transmit a unit of workload from the head node to a processing node.

III. ALGORITHM

In this section, we present our new algorithm for scheduling real-time divisible loads with advance reservation in clusters.

Due to their special 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) [6]. Second, it decides the number 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 algorithms [15], [16], [17], 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 partitions each task and dispatches subtasks to execute on processing nodes.

The real-time divisible load scheduling has been investigated in [8], [9], [6], [10], [7], [13]. However, they do not support advance reservations. A multi-stage real-time scheduling algorithm in [12] supports advance reservations. However, in order to perform the schedulability test, the algorithm 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 this algorithm, 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 this algorithm. First, to make an admission control decision, it reschedules tasks in TWQ. Second, it calculates 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.

In our previous work [13], we investigated the overhead of the brute-force approach and proposed an efficient algorithm, which is linear in the number of tasks in queue or the number of processors in the cluster. It 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 idle waiting times.

However, the algorithm in [13] does not support advance reservations. To address the deficiency of the existing approach, in this paper, we develop a new scheduling algorithm that supports advance reservations. The proposed algorithm can schedule the mixed tasks of reservation and regular (arbitrarily divisible) tasks. The challenges are: 1) how to maintain the linear complexity of the admission controller but also support advance reservations; 2) how to complete all accepted reservations and regular tasks before their deadlines while using a dispatcher that applies a different node allocation (MNA) policy from the policy adopted by the admission controller.

In the following, we describe the admission controller in detail.

A. Admission Control Algorithm

When a task arrives, the scheduler dynamically determines if it is feasible to schedule the new task without compromising the guarantees for previously admitted tasks. According to the newly arrived task’s type, the scheduler invokes an admission test. For a reservation, it first checks if enough resources are available to satisfy the newly arrived task’s type, the scheduler invokes an admission control decision. The ANA is, how inconvenient. 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 idle waiting times.

However, the algorithm in [13] does not support advance reservations. To address the deficiency of the existing approach, in this paper, we develop a new scheduling algorithm that supports advance reservations. The proposed algorithm can schedule the mixed tasks of reservation and regular (arbitrarily divisible) tasks. The challenges are: 1) how to maintain the linear complexity of the admission controller but also support advance reservations; 2) how to complete all accepted reservations and regular tasks before their deadlines while using a dispatcher that applies a different node allocation (MNA) policy from the policy adopted by the admission controller.

In the following, we describe the admission controller in detail.

A. Admission Control Algorithm

When a task arrives, the scheduler dynamically determines if it is feasible to schedule the new task without compromising the guarantees for previously admitted tasks. According to the newly arrived task’s type, the scheduler invokes an admission test. For a reservation, it first checks if enough processing nodes are available to accommodate the reservation. Because data transmission does not happen in parallel, we must then ensure that the new reservation R will not cause any IO conflict. If the admission test is successful, it proves that accepting R will not compromise the guarantees for previously accepted reservations. Admission controller then verifies if accepting the reservation compromises the guarantees to the accepted regular tasks. Accepted reservations are inserted into the reservation queue ResvQueue. For a reservation task R, the reservation duration is provided by the user. The admission controller assumes ANA policy for both reservations and regular tasks. To comply with the ANA admission controller, we convert the reservation task (R_a, R_s, n_r, R_e, IO_ratio) to an all-node scenario (R_a, R_s, N, R_e, IO_ratio) using Eq(3), and insert them into ResvQueue.

\[ R' = (R_e - R_s) * IO\text{ ratio } + \frac{(R_e - R_s) * n_r}{N} \]  

The admission controller only considers the ResvQueue as the reservation queue, while dispatcher dispatches the reservation tasks from ResvQueue.

Since the reservation task admission does not involve node computation and the execution time is fixed, its admission control is kept same as that in [12]. We only modify the admission control algorithm for regular tasks. When a new regular task τ arrives, the algorithm first checks if the head node P_0 will be available early enough to at least finish τ’s data transmission before τ’s absolute deadline. If not so, task τ is rejected. As the next step, task τ is tentatively inserted into TWQ following EDF order and τ’s two adjacent tasks τ_a and τ_p (i.e., the succeeding and the preceding tasks) are identified. By using the information recorded with τ_a and τ_p, the algorithm further tests the schedulability. First, to check whether accepting τ will violate the deadline of any admitted task, the algorithm compares τ’s execution time τ.E with its successor τ_p’s slack_min, which represents the minimum slack of all tasks scheduled after τ. Next, we give the formal definition of slack_min. A task’s slack is defined as,

\[ slack = A + D - (S + E), \]

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.

However, the algorithm in [13] does not support advance reservations. In this paper, we extend the work in [13] to support advance reservations. The new admission controller assumes an ANA policy. We use E and C to respectively denote the task execution time and the task completion time. The admission controller partitions each task following the divisible load theory (DLT), which states that the optimal execution time is obtained when all nodes allocated to a task complete their computation at the same time [14]. Applying this optimal partitioning, we get the execution time of running a task τ(A, σ, D) on N processing nodes as [7],

\[ E(σ, N) = \frac{1 - β}{1 - β N} \sigma (C_{ms} + C_{ps}), \]  

where \[ β = \frac{C_{ps}}{C_{ms} + C_{ps}}. \]  

For a reservation task R, the reservation duration is provided by the user. The admission controller assumes ANA policy for both reservations and regular tasks. To comply with the ANA admission controller, we convert the reservation task (R_a, R_s, n_r, R_e, IO_ratio) to an all-node scenario (R_a, R_s, N, R_e, IO_ratio) using Eq(3), and insert them into ResvQueue.

\[ R' = (R_e - R_s) * IO\text{ ratio } + \frac{(R_e - R_s) * n_r}{N} \]  

The admission controller only considers the ResvQueue as the reservation queue, while dispatcher dispatches the reservation tasks from ResvQueue.

Since the reservation task admission does not involve node computation and the execution time is fixed, its admission control is kept same as that in [12]. We only modify the admission control algorithm for regular tasks. When a new regular task τ arrives, the algorithm first checks if the head node P_0 will be available early enough to at least finish τ’s data transmission before τ’s absolute deadline. If not so, task τ is rejected. As the next step, task τ is tentatively inserted into TWQ following EDF order and τ’s two adjacent tasks τ_a and τ_p (i.e., the succeeding and the preceding tasks) are identified. By using the information recorded with τ_a and τ_p, the algorithm further tests the schedulability. First, to check whether accepting τ will violate the deadline of any admitted task, the algorithm compares τ’s execution time τ.E with its successor τ_p’s slack_min, which represents the minimum slack of all tasks scheduled after τ. Next, we give the formal definition of slack_min. A task’s slack is defined as,

\[ slack = A + D - (S + E), \]  

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.
which reflects the scheduling flexibility of a task. In the equation, \( S \) denotes the task start time. If one or more advance reservations occur in the interval \([A,A+D]\) of a task, the converted all-node execution times (Eq(3)) of the reservations are subtracted from the task’s slack.

If we use \( \text{Sum}_t \) to denote the sum of reservation execution time in the interval \([0, t]\) for the \( \text{ResvQueue} \), the sum of the reservation execution times in the interval \([A,A+D]\) is \( \text{Sum}_{A+D} - \text{Sum}_A \). The array \( \text{Sum}_t \) is updated when a new reservation is accepted, using the all-node model \( R' \) of the reservation \( R \).

\[
\begin{align*}
\text{Sum}_t &= \text{Sum}_t + 1 \\
\text{Sum}_t &= \text{Sum}_t + (R'_e - R'_s) \\
\text{Sum}_t &= \text{Sum}_t + (R'_e - t) \\
\end{align*}
\]

Therefore, we can compute the regular slack time within constant time \( O(1) \). That is:

\[
\text{slack} = A + D - (S + E) - (\text{Sum}_{A+D} - \text{Sum}_A), \quad (5)
\]

Starting a task slack time units later does not violate its deadline. Therefore, as long as \( \tau \)'s execution time is no more than the slack of any preceding 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,

\[
\text{slack}_{min} = \min(\tau_1, \text{slack}; \tau_{i+1}, \text{slack}; \cdots; \tau_n, \text{slack}) \quad (6)
\]

If \( \tau \)'s execution time is less than its successor \( \tau_s \)'s slack, accepting \( \tau \) will not violate any task’s deadline.

The algorithm then checks if task \( \tau \)'s deadline can be satisfied or not. That is, to check if \( \tau(A + D - S) \geq \tau.E + (\text{Sum}_{A+D} - \text{Sum}_S) \), where the start time \( \tau.S \) is the preceding task's completion time \( \tau_p.C \) or \( \tau 's \) arrival time \( \tau.A \). 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 \( \tau \).

**Time Complexity Analysis.** In our admission control 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. The time complexity of the functions to locate \( \tau_a \) and \( \tau_p \), to update the slack, and to update the sum reservation is linear \( O(n) \). Therefore, the new admission control algorithm that supports advance reservations maintains the linear property as that of [13].

**IV. Conclusion**

In this paper, we address the problem of efficient real-time divisible load scheduling with advance reservation. We present a linear algorithm for real-time divisible load scheduling that supports advance reservations in a cluster. This algorithm is expected to significantly improve the system performance with small scheduling overhead, as well as addressing the under-utilization concerns. We plan to further investigate the advantages and disadvantages of the fast admission control algorithm.

**References**


