Feedback Scheduling of Real-Time Divisible Loads in Clusters

Duc Luong, Jitender Deogun, Steve Goddard
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE 68588
{*dluong, deogun, goddard*}*@cse.unl.edu*

Abstract

Quality of Service (QoS) provisioning for divisible loads in clusters can be enabled using real-time scheduling theory, but is based on an important assumption: that the scheduler knows the execution time of every task in the workload. *Information from production clusters, however, shows that estimated execution times of tasks are often inaccurate. Most of the work on scheduling divisible loads on clusters is based on this information, and therefore maybe of limited use when applied in practice. In this paper, we present our ongoing work to develop an EDF (earliest deadline first) scheduling algorithm with a feedback mechanism that is able to solve this problem. The objective of the new algorithm is to provide QoS provisioning of divisible loads when estimated execution times of tasks are inaccurate.*

1 Introduction

Scheduling of arbitrarily divisible loads represents a problem of great significance for cluster-based research computing facilities such as the U.S. CMS (Compact Muon Solenoid) Tier-2 sites [5]. One of the management goals at the University of Nebraska-Lincoln (UNL) Research Computing Facility (RCF) is to provide a multi-tiered QoS scheduling framework in which applications "*pay*" according to the response time requested for a job [5].

Previous work on Quality of Service (QoS) provisioning for divisible loads in a cluster computing environment, however, is based on an important assumption: *the scheduler needs to know the execution time of every task in the workload in advance*. Scheduling decisions may be inefficient if this information is not accurate. Estimation of task execution time is a hard problem not only in real-time systems but also in general cases [6]. Although much work has been done to improve this estimation, there are always uncertainties in task execution times. In distributed systems, this problem becomes even harder because a task might be executed on multiple processors, and communication time should also be considered [1, 7]. Usually, the estimated task execution time is provided to the scheduler along with other task parameters. In most cases, this estimation is the worst-case task execution time, which is obtained empirically or based on expert knowledge of the task. Users who work with clusters tend to overestimate this value "just in case" their job runs longer.

We studied one year's worth of logs for production jobs submitted to the Red and PrairieFire clusters[1] at the University of Nebraska-Lincoln (UNL). We found that among jobs that finish successfully on both Red and Prairiefire clusters, the average execution times are only 9% and 18% of the estimates respectively. In Table 1, we show the number of overestimated and underestimated jobs. According to the current practice, most of the jobs exceeding their estimated execution times are killed. Log information shows that about 91% of such jobs on PrairieFire and 98% on Red, are killed, though these jobs consist of only 3% to 5% of the total number of jobs in a cluster.

Number of jobs	**Red**	**PrairieFire**
Jobs run longer than estimated	6103	1370
Jobs run less than estimated	188545	26193
Jobs that finish on time	0	0
Jobs that are killed	5963	1240
Total	194648	27563

Table 1. Job statistics from two real clusters

QoS provisioning for divisible loads involves three components: an *admission controller* that decides to accept or reject an incoming task, a *scheduler* that schedules and partitions admitted tasks into subtasks, a *dispatcher* that sends the partitioned subtasks to the processors at their scheduled

[1]Red is a 111 node production-mode LINUX cluster, with each node containing two dual core Opteron 275 processors. PrairieFire is a 128 node production-mode LINUX cluster, with each node containing two (single core) Opteron 248 processors.

times. The scheduler makes decisions based on task parameters, such as execution time and deadline. If a task is admitted, it will be placed into the pending queue as a collection of subtasks and later dispatched by the dispatcher. One problem with this model is that once the schedule for a task (and its subtasks) is set, it is not changed. If nodes become available before the scheduled task start time, they are not used. The cluster processing capability is, therefore, wasted. Another problem is that the scheduler does not know how long a task will run after it runs past its allocated time. So, such tasks are generally killed to enforce the schedule. Task killing is, however, undesirable because the time the cluster spends on killed tasks is completely wasted.

We want to achieve the following goals when designing a real-time divisible load scheduling algorithm when execution times of tasks are different from their estimate. First, unused idle time when task finishes earlier than expected must be utilized, so that the system utilization is increased, and we can accept more tasks. Second, overrun tasks are killed only if necessary, i.e., when they cause other tasks to miss their deadline. Task real-time constraints should be guaranteed as long as their execution times are not underestimated. The new scheduling algorithm will be compared with the previous approaches by using simulations as well as experiments on a real cluster.

2 Task and System Models

To develop our scheduling algorithm, we use the same task and system models adopted in [2, 3, 4].

Task Model. A divisible task T_i is denoted by the tuple $T_i = (A_i, \sigma_i, D_i)$ where A_i is arrival time, σ_i is data size and D_i is relative deadline of the task. A workload consists of a set of independent tasks. A task is arbitrarily divisible, which means it can be partitioned into a set of subtasks, each of which processes a portion of the data. We use the vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ to denote the data distribution of a task where n is the number of processing nodes assigned to such a task, and α_i is the data fraction allocated to the i^{th} subtask, which means $\alpha_i \sigma$ unit of data is assigned to subtask i. We have $0 < \alpha_i \leq 1$ and $\sum_{i=1}^{n}(\alpha_i) = 1$.

System Model. The system consists of a cluster with a head node, denoted P_0, connected to N processing nodes, denoted $P_1, P_2, ..., P_N$, via a switch. Every processing node in the cluster has the same computational capability and the same bandwidth on its link to the head node. We call such a cluster homogenous, as apposed to a heterogenous one where computation and transmission capabilities of processing nodes are different from each other. The head node does not participate into the computation but takes the role of the admission controller, the scheduler and the dispatcher. By assumption, data transmission from the head node cannot be done in parallel. Only one processing node can receive data from the head node at a time.

Applying divisible load theory, transmission and computation time of a task is represented by a linear model. The transmission and computation time of σ data units is given by σCms and σCps. Cms represents the time to transmit a unit of workload from the head node to a processing node. Cps represents the time to compute a unit of workload on a single processing node.

3 Algorithms

3.1 Divisible Load Scheduling with Feedback

To develop our algorithm, we adapt the EDF-DLT algorithm [2]. The primary idea of EDF-DLT is to model a homogeneous cluster as heterogeneous and dispatch subtasks at the estimated available time of a processing node, so that the idle time in a cluster node can be better utilized. Recall that $P_1, P_2, \ldots, P_n$ denote n homogenous processors. Assume node P_i could start processing task T at time r_i, for $i = 1, 2, \ldots n$. We call r_i the available time of P_i. It is either the time P_i is released by a previous task or the time task T arrives, whichever is latest. The n nodes are ordered by their available times: P_1 is the earliest at time r_1 and P_n the latest at time r_n

Let $\mathcal{E}$ denote the task execution time when DLT is applied. Cps_i represents the unit processing cost on node P_i and Cms_i denotes the unit transmission cost. Then, as shown in [2], for the heterogeneous model, we have the following,

$$\begin{aligned} Cps_i &= \frac{\mathcal{E}}{\mathcal{E} + r_n - r_i} Cps \qquad (1) \\ Cms_i &= Cms. \qquad (2) \end{aligned}$$

Tasks in a workload have the same Cms and Cps values, which are the estimated time to transmit and compute a single data unit of a task. The actual values, however, may differ from the estimated values.

When a task T_i arrives, the scheduler calculates the minimum number of nodes to be assigned to T_i so that it does not miss its deadline. As shown in [2], the execution time of a task, denoted by $\hat{\mathcal{E}}$, is given by Equation (3),

$$\hat{\mathcal{E}}(\sigma, n) = \sigma Cms + \frac{\prod_{j=2}^{n} X_j}{1 + \sum_{i=2}^{n} \prod_{j=2}^{i} X_j} \sigma Cps \qquad (3)$$

where

$$X_i = \frac{Cps_{i-1}}{Cms + Cps_i}, \text{ for } i = 2, 3 \ldots, n \qquad (4)$$

and the minimum number of nodes assigned to a task is given by:

$$\tilde{n}^{min} = \lceil \frac{\ln \gamma}{\ln \beta} \rceil \qquad (5)$$

where

$$\gamma = 1 - \frac{\sigma C_{ms}}{A + D - r_n} \tag{6}$$

and

$$\beta = \frac{Cps}{Cms + Cps}. \tag{7}$$

The data distribution vector is given as

$$\sigma_1 = \frac{\sigma}{1 + \sum_{i=2}^{n} \prod_{j=2}^{i} X_j} \tag{8}$$

and,

$$\sigma_i \;=\; \frac{\prod_{j=2}^{i} X_j \sigma}{1 + \sum_{i=2}^{n} \prod_{j=2}^{i} X_j}, \text{ for } i = 2, 3 \ldots, n \tag{9}$$

The results from [2] show that EDF-DLT is one of the best known scheduling algorithms for real-time divisible loads in clusters. This algorithm assumes that the estimate of task execution time is correct. However, if the actual values of Cms and Cps do not match the user's estimate, tasks would either finish earlier or run past their estimated execution time. Since there is no feedback mechanism incorporated in the above algorithms, the scheduler has no means of knowing about these situations. This leads to idle time that is not utilized or tasks being killed because their allocated time expires.

We propose DLSwF, a DLT-based scheduling algorithm with a feedback mechanism, to handle these cases. Its goal is to better utilize the processing nodes and minimize the number of tasks that are killed. We use the following definitions to describe how DLSwF works:

- A task is said to *"underrun"* if its execution time is smaller than the estimated value. Most of the tasks on real clusters fall into this category. A task that underruns is called an underrun task.
- A task is said to *"overrun"* if its execution time is larger than the estimated value. A task that overruns is called an overrun task.

The general process of the DLSwF algorithm is shown in Pseudocode 1. It is based on four events in the system. The NewTaskEvent is invoked when a task arrives. We use the function Admission Control to check if we can accept the task or not. If it is accepted, this function generates the data distribution and the schedule for the task.

Due to the feedback module, the system is able to detect and handle the two events: OverrunTimerEvent and TerminationEvent. The first event is invoked when a subtask does not finish at its expected completion time. The second event is invoked when a subtask finishes its execution. The mechanism to handle these two events are described in Section 3.2.

Pseudocode 1 DLSwF(Event)

```
 1: if Event is NewTaskEvent then
 2:     call AdmissionControl to decide whether the task can be
        admitted or not
 3:     call GenerateSchedule to partition the task if it is admitted
 4: else if Event is OverrunTimerEvent then
 5:     handle overrun and update nodes status
 6: else if Event is TerminationEvent then
 7:     update nodes status
 8: else if Event is DispatchTimerEvent then
 9:     //this event is handled by DispatchTask()
10: end if
11: call DispatchTask()
12: return
```

The DispatchTimerEvent is invoked when a subtask in the dispatching queue to be submitted.

After processing any of these events, the system invokes the DispatchTask function. This function is to dispatch a subtask in the dispatching queue, if any, to a processing node in the cluster. After dispatching a subtask, it will reset the DispatchTimer to the time when the next subtask should be submitted.

3.2 Handling Overrun and Underrun Tasks

Since the scheduler is not clairvoyant, it cannot know if a task underruns/overruns until its subtasks finish. Therefore, if a task overruns, it will be difficult for the scheduler to estimate the termination time of such a task in order to schedule the next tasks correctly. The nodes occupied by overrun tasks are considered to be blocked, or to have estimated finish times at ∞. An overrun task can therefore severely affect the acceptance of new tasks and result in accepted tasks missing their deadlines.

Common practice on real clusters is to kill overrun tasks, the EDF-DLT algorithm also uses such an approach to ensure overrun tasks do not cause other tasks to miss deadlines or new tasks to be rejected. However, killing an overrun task is costly because the time the system has spent on that task is wasted and the task would have to be resubmitted later. Thus, our algorithm tries not to kill overrun tasks if it is avoidable. Still, deadlines of tasks that do not overrun should not be missed.

In the DLSwF algorithm, an overrun task is allowed to continue to run as long as it does not: (i) cause any already accepted task to miss its deadline or (ii) prevent a new task from being accepted.

Condition (i) says that when a task overruns, it should not cause any other tasks to miss their deadline, otherwise, the overrun tasks will be killed. Condition (ii) says that if a new task can only be accepted with the nodes occupied by the overrun tasks then overrun tasks will be killed. Intuitively, this method works well in the case where the system is not heavily loaded. But when the system is very busy, the algorithm cannot prevent overrun tasks from being killed. If the two conditions are enforced, an admitted task will not miss its deadline unless it overruns.

The HandleOverrun function is described as follows. Assume that an overrun task occurs at time t, we need to gather the following information in order to handle the situation:

N_{OR}: Number of nodes that have an overrun subtask.

D_T: Number of subtasks waiting to be dispatched at time t.

N_{AV}: Number of available nodes at time t.

It may be noted that $N_{OR} > 0$, $D_T \geq 0$ and $N_{AV} \geq 0$, since it is assumed that at least one overrun task exists.

Based on D_T and N_{AV}, we evaluate the available time t' of blocked nodes to ensure that the schedule is being enforced. In other words, we need to determine when these nodes must finish their jobs. There are two cases:

- Case 1: $0 \leq D_T \leq N_{AV}$.
  In this case, there are subtasks that must be dispatched at time t, and sufficient nodes are available. Therefore overrun tasks can continue to execute.

- Case 2: $D_T > N_{AV}$.
  In this case, a sufficient number of nodes are not available. However, we see that all subtasks do not start at the same time and thus some have to wait until others finish their data transmission. Therefore, if we order the subtasks in increasing order of their start time, we can let the overrun jobs continue to run until the k^{th} subtask starts, with $k = D_T - N_{AV}$.

As opposed to the overrun case, the solution for underrun tasks is relatively straightforward. The system knows immediately when a task underruns because of the feedback mechanism, i.e., TerminationEvent is detected before the expected completion time of a task. Therefore, it is able to update nodes status and if there is a pending task in the dispatching queue, this task will be dispatched immediately.

4 Conclusions and Future Work

In this paper, we address the problem of inaccuracy in the estimated execution times in the context of real-time divisible load scheduling. We present an approach to identify and handle overrun and underrun tasks. QoS and real-time constraints of the system are enforced by integrating the feedback mechanism into the scheduling algorithm. Our algorithm is expected to significantly improve the system performance with different levels of uncertainty in tasks execution time. We plan to consider the following issues when developing the algorithm: (i) applying historical knowledge of the workload to improve the admission control of the scheduler and (ii) detecting failure nodes in the cluster and reconfiguring the scheduler when nodes are added/removed from the cluster.

References

[1] M. Drozdowski. Estimating execution time of distributed applications. In *Proceedings of the Parallel Processing and Applied Mathematics : 4th International Conference, PPAM 2001 Naleczow, Poland, September 9-12, 2001. Revised Paper*, pages 593–596. Springer Berlin / Heidelberg, 2002.

[2] X. Lin, Y. Lu, J. Deogun, and S. Goddard. Real-time divisible load scheduling with different processor available times. In *Proceedings of the 2007 International Conference on Parallel Processing (ICPP 2007).*

[3] X. Lin, Y. Lu, J. Deogun, and S. Goddard. Enhanced real-time divisible load scheduling with different processor available times. In *14th International Conference on High Performance Computing*, December 2007.

[4] X. Lin, Y. Lu, J. Deogun, and S. Goddard. Real-time divisible load scheduling for cluster computing. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Application Symposium*, pages 303–314, Bellevue, WA, April 2007.

[5] D. Swanson. Personal communication. Director, UNL Research Computing Facility (RCF) and UNL CMS Tier-2 Site, August 2005.

[6] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckman, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution time problem - overview of methods and survey of tools. In *ACM Transactions on Embedded Computing Systems (Accepted January 2007).*

[7] C.-T. Yang, P.-C. Shih, C.-F. Lin, C.-H. Hsu, and K.-C. Li. A chronological history-based execution time estimation model for embarrassingly parallel applications on grids. In *Proceedings of the Parallel and Distributed Processing and Applications*, pages 425–430. Springer Berlin / Heidelberg, 2005.