Real-time scheduling of divisible loads in cluster computing environments*

Xuan Lin, Anwar Mamat, Ying Lu*, Jitender Deogun, Steve Goddard
University of Nebraska, Lincoln, United States

A R T I C L E   I N F O

Article history:
Received 17 November 2008
Received in revised form
7 October 2009
Accepted 26 November 2009
Available online 4 December 2009

Keywords:
Real-time scheduling
Parallel computing
Divisible load
QoS
Cluster computing

A B S T R A C T

Cluster computing has become an important paradigm for solving large-scale problems. To enhance the quality of service (QoS) and provide performance guarantees in a cluster computing environment, various real-time scheduling algorithms and workload models have been investigated. Computational loads that can be arbitrarily divided into independent tasks represent many real-world applications. However, the problem of providing performance guarantees to divisible load applications has only recently been studied systematically. In this work, three important and necessary design decisions, (1) workload partitioning, (2) node assignment, and (3) task execution order, are identified for real-time divisible load scheduling. A scheduling framework that can configure different policies for each of the three design decisions is proposed and used to generate various algorithms. This paper systematically studies these algorithms and identifies scenarios where the choices of design parameters have significant effects.

© 2009 Elsevier Inc. All rights reserved.

1. Introduction

Cluster computing has become an important paradigm for solving large-scale problems. However, as the size of a cluster increases, so does the complexity of resource management and maintenance. Therefore, automated performance control and resource management are expected to play critical roles in sustaining the evolution of cluster computing. The current cluster scheduling practice is similar in sophistication to early supercomputer batch scheduling algorithms, and no consideration is given to desired quality-of-service (QoS) attributes. To fully take advantage of the power of computational clusters, new scheduling theory that provides high performance, QoS assurance, and streamlined management of the cluster resources needs to be developed.

Real-time scheduling theory has been very successful in providing deterministic QoS in desktop systems [10,11,25]. A significant challenge in developing real-time scheduling theory for cluster computing, however, is to support various types of cluster applications. Broadly speaking, computational loads submitted to a cluster can be structured in three primary ways: indivisible, modularly divisible, and arbitrarily divisible. An indivisible load is essentially a sequential job which cannot be further divided, and thus must be assigned to a single processor. Modularly divisible loads can be divided a priori into a certain number of subtasks and are often described by a task (or processing) graph. Arbitrarily divisible loads, also called embarrassingly parallel workloads, can be partitioned into an arbitrarily large number of independent load fractions. Examples of arbitrarily divisible loads can be easily found in high-energy and particle physics as well as biometrics. For example, the CMS (Compact Muon Solenoid) [15] and ATLAS (A Toroidal LHC Apparatus) [6] projects, which are associated with the LHC (Large Hadron Collider) at CERN (European Laboratory for Particle Physics), execute cluster-based applications with arbitrarily divisible loads. Usually all elements in such computational loads demand an identical type of processing, and relative to the huge total computation, the processing on each individual element is infinitesimally small. The problem of providing QoS or real-time guarantees for sequential and modularly divisible jobs in distributed systems has been studied extensively. However, despite the increasing importance of arbitrarily divisible applications [33], to the best of our knowledge, the real-time scheduling of arbitrarily divisible loads has not been systematically investigated.

Scheduling of arbitrarily divisible loads represents a problem of great significance for cluster-based research computing facilities such as the US CMS Tier-2 sites [37]. For example, one of the management goals at the University of Nebraska–Lincoln (UNL) Research Computing Facility (RCF) (a CMS Tier-2 site) is to provide a multi-tiered QoS scheduling framework in which applications “pay” according to the response time requested for each job [37]. Existing real-time cluster scheduling algorithms assume the existence of a task graph for all applications, which are not appropriate for arbitrarily divisible loads. To better manage these high-end clusters and control their performance, we need new real-time scheduling algorithms for arbitrarily divisible applications.

* This work is funded by NSF grant No. 0720810.
* Corresponding address: Department of Computer Science and Engineering, University of Nebraska, Lincoln, United States.
E-mail addresses: ylu@cse.unl.edu (Y. Lu), goddard@cse.unl.edu (S. Goddard),
Three contributions are made in this paper. First, three important and necessary design decisions for real-time divisible load scheduling are identified. Second, a scheduling framework that integrates the three components is proposed. By configuring different policies for each of these three decisions, various real-time scheduling algorithms can be generated. Third, we systematically investigate the effects of these design decisions and system parameters on a set of real-time scheduling algorithms. Analysis and experimental results demonstrate a need for adaptive scheduling approaches.

The remainder of this paper is organized as follows. Section 2 describes both task and system models. In Section 3, the real-time scheduling algorithms investigated in this paper are discussed. Detailed analysis of design decisions are given in Sections 4 and 5. We evaluate the performance of algorithms in Section 6 and present the related work in Section 7. Section 8 concludes the paper.

2. Task and system models

In this section we describe our task and system models briefly, and state assumptions related to these models.

**Task model.** We assume a real-time aperiodic task model in which each aperiodic task \( T_i \) consists of a single invocation specified by the tuple \((A_i, \sigma_i, D_i)\), where \( A_i \) is the task arrival time, \( \sigma_i \) is the total data size of the task, and \( D_i \) is the task relative deadline. The task absolute deadline is given by \( A_i + D_i \). Section 4 presents, in detail, how the task execution time is dynamically computed based on the total data size \( \sigma_i \), allocated resources (i.e., processing nodes and bandwidth) and the partitioning method applied to parallelize the computation. There are many applications [17] conforming to this divisible load task model: for example, distributed search for a pattern in text, audio, graphical, and database files; distributed processing of big measurement data files; and many simulation problems. For such applications, we often derive their data sizes \( \sigma_i \) based on their input file sizes.

**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 happen in parallel, although it is straightforward to generalize our model and include cases where some pipelining of communication may occur. For arbitrarily divisible loads, tasks and subtasks are independent. Therefore, when executing such applications processing nodes do not communicate with each other.

3. Algorithms

This section presents real-time scheduling algorithms for divisible loads. To develop the algorithms, we need to make three important decisions. The first is to adopt a scheduling policy to determine the order of execution for tasks (Section 3.1). The second decision is to choose a strategy to partition the task (Section 3.2); that is, to partition the task data among a given number of computing resources. The last decision is to determine the number \( n \) of processing nodes to assign to each task (Section 3.3). Basically, for a real-time divisible task, the number of the processing nodes assigned to it can be between the minimum number of nodes for it to complete before its deadline and all available processing nodes in the system.

3.1. Scheduling policies

Three scheduling policies to determine the execution order of tasks are investigated: FIFO (First In First Out), EDF (Earliest Deadline First) and MWF (Maximum Workload derivative First) [23]. The FIFO scheduling algorithm executes tasks following their order of arrival and is a common practice adopted by cluster administrators to manage a task queue. EDF, a well-known real-time scheduling algorithm, orders tasks by their absolute deadlines. As a real-time scheduling algorithm for divisible tasks, the main rules of MWF [23] are: (1) the task with the highest workload derivative \( \delta w_i \) is scheduled first; and (2) the number of nodes allocated to a task is kept as small as possible \( (n_i^{\text{min}}) \) without violating its deadline. Node assignment is described in Section 3.3. Here, we review how MWF determines the task execution order and defines the workload derivative metric, \( \delta w_i \).

\[
\delta w_i = w_i(n_i^{\text{min}} + 1) - w_i(n_i^{\text{min}}),
\]

where \( w_i(n) \) denotes the workload (cost) of a task \( T_i \) when \( n \) processing nodes are assigned to it. That is, \( w_i(n) = n \times \sigma_i(n, n) \), where \( \sigma_i(n, n) \) denotes the task execution time (see Section 4 for \( \sigma \)’s calculation). Therefore, \( \delta w_i \) is the derivative of the task workload \( w_i(n) \) at \( n^{\text{min}} \) (the minimum number of nodes needed by \( T_i \) to meet its deadline).

3.2. Task partitioning methods

We apply a task partitioning method to divide a task among its allocated processing nodes. Two different partitioning methods are investigated: the Optimal Partitioning Rule (OPR), and the Equal Partitioning Rule (EPR). The OPR is based on 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 [39]. For comparison, we propose the EPR, based on a common practice of dividing a task into \( n \) equal-sized subtasks when the task is to be processed by \( n \) nodes. When different partitioning methods are applied to parallelize a task’s computation, the task will experience different execution time and may require varied minimum number \( n_i^{\text{min}} \) of nodes. In Section 4, we provide detailed analysis on these partitioning methods and derive the task execution time and \( n_i^{\text{min}} \) for each of them.

3.3. Node assignment policies

Node assignment determines the number of processing nodes allocated to a task. In this paper, we study two primary strategies for node assignment. First, assign a task all \( N \) or \( n^* \) (i.e., min\((N, n^*)\)) nodes to finish it as early as possible (see Section 5 for \( n^* \)’s description). Second, assign a task the minimum number \( n_i^{\text{min}} \) of nodes it needs to meet its deadline and thereby save resources for new tasks. To guarantee that a task finishes by its deadline, the real-time scheduler must know the minimum number of nodes required by the task. Since \( n_i^{\text{min}} \) is determined by not only the task data size, deadline and execution start time but also the applied partitioning method, we derive \( n_i^{\text{min}} \) in Section 4 when the partitioning methods are thoroughly analyzed.

3.4. Algorithm framework

As is typical for dynamic real-time scheduling algorithms [31, 16, 26], when a new task arrives, the scheduler dynamically determines if it is feasible to schedule the task without compromising the guarantees for previously admitted tasks. The general framework for a schedulability test is shown in Fig. 1. It can be configured to generate various real-time divisible load scheduling algorithms by giving the design decisions on: (1) scheduling policy (FIFO, EDF or MWF), (2) task partitioning rule (OPR or EPR), and (3) node assignment method (assigning a task \( \text{min}(N, n^*) \) or \( n_i^{\text{min}} \) nodes). Upon completion of the test, if all tasks are schedulable a feasible
Data Structure:
- \( n^{\text{min}}(t) \): the minimum number of processing nodes needed to finish \( T_i \) before its deadline, assuming it is dispatched at time \( t \).
- AvailableNodelist < \( l_k, A_N_k > \): a list of the number of available nodes along with the time, where \( l_k \) is the time and \( A_N_k \) is the number of available nodes.

Pseudocode:

```java
boolean Schedulability-Test(T)
{
    TempTaskList ← T + AdmitTasksQueue
    order TempTaskList /× EDF, FIFO or MWF (Decision 1) /
    generate AvailableNodelist /× Obtain the available nodes information /
    SchedulableList ← ∅ /× Initialization /
    while TempTaskList ! = ∅
    {
        /× Trying to assign a task \( n^{\text{min}} \) or \( \min(N, n^{\text{min}}) \) nodes (Decision 3)/
        identify the first task \( T_k \) and the earliest time \( l_k \) where the available nodes \( A_N_k \geq n^{\text{min}}(l_k) \) or identify the earliest time \( l_k \) when \( A_N_k \geq N \)
        remove \( T_k(\alpha_1, \sigma, D) \) from TempTaskList
        \( s_k \leftarrow l_k \) /× Set the scheduled starting time /
        \( n_k \leftarrow n^{\text{min}}(l_k) \) or \( n_k \leftarrow \min(N, n^{\text{min}}(l_k)) \)
        /× According to the chosen partitioning rule: OPR or EPR (Decision 2), set the expected completion time following Eq. 8 or Eq. 19 /
        \( c_k \leftarrow C_{(\alpha_1, n_k)} + s_k \)
        if \( c_k \geq D_k \)
            return false /× Deadline misses /
        put \( T_k(\alpha_1, \sigma, D, s_k, n_k, c_k) \) into SchedulableList
        update AvailableNodelist
    }
    end while
    /× All tasks in the cluster are schedulable /
    AdmitTasksQueue ← SchedulableList
    return true
}
```

Fig. 1. Schedulability test for the algorithms.

4. Analysis of task partitioning methods

Section 3.2 has introduced the two partitioning methods that we will investigate in this paper. In this section, we analyze these methods in detail. Since different partitioning methods lead to different task executions, we derive the task execution time and \( n^{\text{min}} \) for each of these methods. These analysis provide essential ingredients for the real-time scheduling algorithms (Fig. 1).

The following notations, partially adopted from [39], are used in the analysis.
- \( T = (A, \sigma, D) \): A divisible task, where \( A \) is the arrival time, \( \sigma \) is the data size, and \( D \) is the relative deadline.
- \( \alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n) \): Data distribution vector, where \( n \) is the number of processing nodes allocated to the task, \( \alpha_i \) is the data fraction allocated to the \( i \)-th node, i.e., \( \alpha_i \sigma \) is the amount of data that is to be transmitted to the \( i \)-th node for processing, \( 0 < \alpha_i \leq 1 \) and \( \sum_{i=1}^{n} \alpha_i = 1 \).
- \( \tau \): Cost of transmitting a unit workload.
- \( \chi \): Cost of processing a unit workload.
- \( \theta_{cm} \): The setup time (cost) for the head node to initialize a communication on a link.
- \( \theta_{cp} \): The setup time (cost) for a processing node to initialize a computation.

In the analysis, depending on whether the task setup costs (i.e., \( \theta_{cm} \) and \( \theta_{cp} \)) are negligible or not, we have two different scenarios. Similar to the previous work on divisible loads [39], linear models are used to represent processing and transmission times. When the setup costs are negligible, the data transmission (or communication) time on the \( j \)-th link is \( C_{ij}(\alpha_i \sigma) = \alpha_i \sigma \), and the data processing time on the \( j \)-th node is \( C_p(\alpha_i \sigma) = \alpha_i \sigma \chi \); and when the setup costs are significant, the data transmission and processing costs are \( \theta_{cm} + \alpha_i \sigma \tau \) and \( \theta_{cp} + \alpha_i \sigma \chi \), respectively. In the following four sections, scenarios both with or without setup costs are analyzed for the two partitioning methods (i.e., OPR and EPR). To analyze the task execution time for the OPR, a method proposed by Bharadwaj et al. [9] is adopted.

4.1. Optimal partitioning rule (OPR), no setup costs

For a given task, let \( \delta \) denote the Task Execution Time, which is a function of \( \sigma \) and \( n \). We first analyze the execution time function, \( \delta(\sigma, n) \), assuming that \( n \) nodes are to be allocated to process a total data size of \( \sigma \). Then, we use it to derive the minimum number, \( n^{\text{min}} \), of nodes needed to meet the task deadline.

\[ T = (A, \sigma, D); A \text{ divisible task, } A \text{ arrival time, } \sigma \text{ data size, } D \text{ relative deadline} \]
\[ \alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n); \text{ data distribution vector, } n \text{ number of nodes allocated, } \alpha_i \text{ fraction to node } i \]
\[ \delta(\sigma, n) \text{ function of } \sigma \text{ and } n \text{ to process } \sigma \]
\[ \delta \text{ depends on } \sigma \text{ and } n \text{ for OPR, no setup costs} \]
(1-a) Task execution time analysis: Fig. 2(a) shows an example task execution time diagram when the setup costs are negligible. The OPR is followed to partition the task among \( n \) processing nodes so that all nodes complete their computation of the task at the same time. We have

\[
\mathbb{E}(\sigma, n) = \alpha_1 \sigma \tau + \alpha_1 \sigma \chi
\]

\( (2) \)

\[
= (\alpha_1 + \alpha_2) \sigma \tau + \alpha_2 \sigma \chi
\]

\( (3) \)

\[
= (\alpha_1 + \alpha_2 + \alpha_3) \sigma \tau + \alpha_3 \sigma \chi
\]

\( (4) \)

\[
\ldots
= (\alpha_1 + \alpha_2 + \alpha_3 + \cdots + \alpha_n) \sigma \tau + \alpha_n \sigma \chi.
\]

From Eqs. (2) and (3), we get

\[
\alpha_2 = \alpha_1 \frac{\sigma \chi}{\sigma \tau + \sigma \chi}
\]

\( (6) \)

Let \( \beta = \frac{\chi}{\tau + \chi} \), we have

\[
\alpha_2 = \beta \alpha_1.
\]

Similarly, from Eqs. (3) and (4), we have \( \alpha_3 = \beta^2 \alpha_1 \), and therefore, \( \alpha_3 = \beta^2 \alpha_1 \). This leads to a general formula: \( \alpha_j = \beta^{j-1} \alpha_1 \). Since \( \alpha_j \) is the data fraction distributed to the \( j \)th processing node, we have \( \sum_{j=1}^{n} \alpha_j = 1 \). Substituting \( \alpha_j \) with \( \beta^{j-1} \alpha_1 \) in this equation, we obtain

\[
\alpha_1 + \beta \alpha_1 + \beta^2 \alpha_1 + \cdots + \beta^{n-1} \alpha_1 = 1.
\]

Solving this equation, we get \( \alpha_1 = \frac{1 - \beta}{1 - \beta^n} \). Substituting it into Eq. (2), we have

\[
\mathbb{E}(\sigma, n) = \frac{1 - \beta}{1 - \beta^n} \sigma (\tau + \chi).
\]

(8)

(1-b) Derivation of \( n^\text{min} \): Given \( \mathbb{E}(\sigma, n) \), we can calculate the minimum number of \( n^\text{min} \) nodes required to meet a task deadline.

Let \( C(n) \) denote the task completion time function. Assuming that the task \( T = (A, \sigma, D) \) has a start time \( s \), then its completion time is \( C(n) = s + \mathbb{E}(\sigma, n) \), which leads to

\[
C(n) = s + \frac{1 - \beta}{1 - \beta^n} \sigma (\tau + \chi).
\]

(9)

To meet a task deadline means its completion time must satisfy the constraint that \( C(n) \leq A + D \). It follows that

\[
s + \frac{1 - \beta}{1 - \beta^n} \sigma (\tau + \chi) \leq A + D,
\]

that is

\[
1 - \beta \leq \frac{1 - \beta^n}{1 - \beta} \sigma (\tau + \chi) \leq A + D - s.
\]

(10)

Since \( 0 < \beta < 1 \), we have \( 1 - \beta^n > 0 \). Multiplying both sides of Eq. (10) by \( 1 - \beta^n \), we get

\[
1 - \beta \sigma (\tau + \chi) \leq (1 - \beta^n) (A + D - s).
\]

(11)

If \( A + D - s \leq 0 \), the task will miss its deadline no matter how many nodes are assigned to it and how we partition it. Such a task will be rejected because it fails the schedulability test (Section 3.4) of our algorithms. Thus, \( A + D - s > 0 \), and dividing both sides of Eq. (11) by \( (A + D - s) \) we have

\[
1 - \beta^n \geq \frac{(1 - \beta) \sigma (\tau + \chi)}{A + D - s} \quad \text{thus}
\]

\[
\beta^n \leq 1 - \frac{(1 - \beta) \sigma (\tau + \chi)}{A + D - s} = 1 - \frac{\sigma (\tau + \chi)}{A + D - s}.
\]

Let \( \gamma = 1 - \frac{\sigma (\tau + \chi)}{A + D - s} \). Thus, \( \beta^n \leq \gamma \) is required. If \( \gamma \leq 0 \), starting task \( T \) at time \( s \) will not leave enough time even for its data transmission, and therefore the task will be rejected. Thus, \( \gamma > 0 \), and it follows that \( n \geq \frac{\ln \gamma}{\ln \beta} \). Since \( n \), the number of nodes assigned, should be an integer, we have \( n \geq \lceil \frac{\ln \gamma}{\ln \beta} \rceil \). Therefore, the minimum number of processing nodes that the task needs at time \( s \) to meet its deadline is \( n^\text{min} = \lceil \frac{\ln \gamma}{\ln \beta} \rceil \), where \( \gamma \) is defined above and \( \beta \) in Eq. (7).

4.2. Optimal Partitioning Rule (OPR) with setup costs

In Section 4.1, we have considered the scenario where the setup costs for initializing data transmission and data processing are negligible. In this section, we present the analysis for the case where the setup costs are significant. The goal of the analysis remains the same: to derive the task execution time function and the minimum number of nodes required to meet a task deadline.

The setup cost of communication comes from physical network latencies, network protocol overhead, or middleware overhead. In the TeraGrid project [38], the network speed can be up to 40 Gbps with a latency of around 100 ms. That is, the latency contributes to about 1/3 of the time required to send 1 GB of data. It has also been shown that the setup cost for computation can be up to 25 s in practice [13], which is significant for small tasks.

(2-a) Task execution time analysis: Taking the setup costs into consideration, the data transmission time on the jth link is modeled as \( C_m(\alpha \sigma) = \theta_{cm} + \alpha \sigma \tau \), and the data processing time on the jth node is \( C_p(\alpha \sigma) = \theta_{cp} + \alpha \sigma \chi \). Fig. 2(b) shows an example task execution time diagram following the OPR when \( n \) nodes are allocated to a task and the setup costs are modeled. Analyzing the time diagram, we derive the Task Execution Time \( \mathbb{E} \) as follows:

\[
\mathbb{E}(\sigma, n) = (\theta_{cm} + \alpha \sigma \tau) + (\theta_{cp} + \alpha \sigma \chi)
\]

(12)

---

1. Rejection in the cluster environment means that the system administrator (or a program proxy) will negotiate with the client for a feasible task deadline, and the job will be rescheduled with modified parameters.
By analyzing the diagram, we have (3-a)

\[ \varepsilon = 2\theta \omega + (\alpha_1 + \alpha_2) \sigma \tau + (\theta \omega + \alpha_2 \sigma \chi) \]  
\[ = 3\theta \omega + (\alpha_1 + \alpha_2 + \alpha_3) \sigma \tau + (\theta \omega + \alpha_3 \sigma \chi) \]  
\[ \cdots \]  
\[ = n\theta \omega + (\alpha_1 + \alpha_2 + \alpha_3 + \cdots + \alpha_n) \sigma \tau + (\theta \omega + \alpha_n \sigma \chi) \]  

From Eqs. (12) and (13), we have \( \omega_2 = \omega_1 \beta - \phi \), where

\[ \beta = \frac{\chi}{\tau + \chi} \quad \text{and} \quad \phi = \frac{\theta_\omega}{\sigma (\tau + \chi)} \]  

Similarly, from Eqs. (13) and (14), we get \( \omega_3 = \omega_1 \beta - \phi \), and therefore \( \omega_1 = \omega_1 \beta^2 - \beta \phi - \phi \), leading to the general formula

\[ \omega_j = \omega_1 \beta^{j-1} - \sum_{k=0}^{j-2} \beta^k \phi. \]

Thus

\[ \omega_j = \omega_1 \beta^{j-1} - \frac{1 - \beta^{j-1}}{1 - \beta} \phi, \quad \text{for } j = 2, 3, \ldots, n. \]

Now, substituting \( \omega_j \) with \( (\alpha_1 \beta^{j-1} - \frac{1 - \beta^{j-1}}{1 - \beta} \phi) \) in the equation

\[ \sum_{j=1}^{n} \omega_j = 1, \]

we get

\[ \alpha_1 + \sum_{j=2}^{n} \left( \alpha_1 \beta^{j-1} - \frac{1 - \beta^{j-1}}{1 - \beta} \phi \right) = 1 \]

i.e.,

\[ \alpha_1 + \sum_{j=1}^{n} \left( \alpha_1 \beta^{j-1} - \frac{1 - \beta^{j-1}}{1 - \beta} \phi \right) = 1. \]

A solution to the above equation leads to

\[ \alpha_1 = \frac{1 - \beta}{1 - \beta^n} + \frac{n \phi}{1 - \beta^n} - \frac{\phi}{1 - \beta}. \]

Let \( \beta(n) = \frac{1 - \beta}{1 - \beta^n} + \frac{n \phi}{1 - \beta^n} - \frac{\phi}{1 - \beta} \).

It follows that

\[ \varepsilon (\sigma, n) = \theta + \omega + \sigma (\tau + \chi) \beta(n). \]  

(2-b) Derivation of \( n_{\text{min}} \): If task \( T = (A, \sigma, D) \) has a start time \( s \), then, to meet its deadline, \( \varepsilon (\sigma, n) \leq A + D - s \) must be satisfied. That is,

\[ \theta + \omega + \sigma (\tau + \chi) \beta(n) \leq A + D - s. \]  

This constraint can be solved numerically. The smallest integer \( n \) that satisfies the constraint is the minimum number \( n_{\text{min}} \) of nodes that need to be assigned to task \( T \) at time \( s \) to meet its deadline.

Note that the model without setup costs (Section 4.1) is a special case of this model, where \( \theta = \omega = 0 \) and accordingly \( \phi = \frac{\sigma \tau}{\sigma (\tau + \chi)} \leq 1 \). Thus, the constraint Eq. (17) reduces to \( \sigma (\tau + \chi) \geq A + D - s \), the constraint Eq. (10) derived in Section 4.1.

4.3. Equal Partitioning Rule (EPR), no setup costs

In this section, we analyze the EPR that partitions a task into equal-sized subtasks. Similar to the analysis for the OPR, we derive the task execution time function and \( n_{\text{min}} \) for the case when the EPR is applied.

(3-a) Task execution time analysis: Assuming that \( n \) nodes are allocated to a task and there is no significant setup cost, an example task execution time diagram following the EPR is shown in Fig. 3(a). By analyzing the diagram, we have \( \varepsilon = \sigma + \frac{\sigma \chi}{n} \). Thus

\[ \varepsilon (\sigma, n) = \sigma + \frac{\sigma \chi}{n}. \]

(3-b) Derivation of \( n_{\text{min}} \): Assuming that the task \( T = (A, \sigma, D) \) has a start time \( s \), then the task completion time \( C(n) = s + \varepsilon(n) \) must satisfy the constraint \( C(n) \leq A + D \). That is,

\[ s + \sigma + \frac{\sigma \chi}{n} \leq A + D. \]  

Thus \( \frac{\sigma \chi}{A + D - s - \sigma} \leq n \). Therefore, following the EPR the minimum number of processing nodes that the task needs at time \( s \) to complete before its deadline is \( n_{\text{min}} = \left\lceil \frac{\sigma \chi}{A + D - s - \sigma} \right\rceil \).

4.4. Equal Partitioning Rule (EPR) with setup costs

In this section, we present the analysis of the EPR when the setup costs are significant.

(4-a) Task execution time analysis: Fig. 3(b) shows an example task execution time diagram following the EPR when \( n \) nodes are allocated to a task and the setup costs are modeled. By analyzing the time diagram, we have \( \varepsilon(n) = n\theta + \sigma \tau + \omega + \sigma \chi \), where \( \omega_n = \frac{1}{n} \). Thus

\[ \varepsilon(n) = n\theta + \sigma \tau + \omega + \sigma \chi. \]

(4-b) Derivation of \( n_{\text{min}} \): Assuming that the task \( T = (A, \sigma, D) \) has a start time \( s \), then the task completion time \( C(n) = s + \varepsilon(n) \) must satisfy the constraint \( C(n) \leq A + D \). That is,

\[ s + n\theta + \sigma \tau + \omega + \sigma \chi \leq A + D. \]

We have

\[ \theta n^2 - \omega n + \sigma \chi \leq 0. \]  

Since \( \theta > 0 \), \( Y = \theta n^2 - \omega n + \sigma \chi \) is a parabola that opens upward. Fig. 4 shows three representations of the parabola, when \( \omega^2 - 4\sigma \chi \theta > 0 \), \( \omega^2 - 4\sigma \chi \theta = 0 \), and \( \omega^2 - 4\sigma \chi \theta < 0 \). Thus, to derive \( n_{\text{min}} \) the three cases need to be considered.

In the first case, when \( \omega^2 - 4\sigma \chi \theta > 0 \), the parabola has no real axis intercepts, which implies that \( Y = \theta n^2 - \omega n + \sigma \chi \) will always be greater than 0. Therefore constraint Eq. (22) cannot be satisfied for any real number \( n \), meaning that it is impossible to meet the task deadline at time \( s \).

In the second case, when \( \omega^2 - 4\sigma \chi \theta = 0 \), the parabola has only one real axis intercept where \( n = \frac{\omega}{2\theta} \). This is the only possible value of \( n \) that satisfies constraint Eq. (22). In addition, \( n \), the number of processing nodes, must be a positive integer. Thus, the task can meet its deadline at time \( s \) if and only if \( n = \frac{\omega}{2\theta} \) is a positive integer.

In the third case, when \( \omega^2 - 4\sigma \chi \theta < 0 \), the parabola has two real axis intercepts. From Fig. 4, we can see that, in order to satisfy constraint equation (22), the value of \( n \) should fall between the two real roots of equation \( \theta n^2 - \omega n + \sigma \chi = 0 \). That is,

\[ n = \frac{-\omega + \sqrt{\omega^2 - 4\theta\omega\sigma\chi}}{2\theta} \]

Since \( n \) must be a positive integer, in this case the maximum number of nodes needed for the task to complete before its deadline is

\[ n_{\text{min}} = \begin{cases} 
\frac{N}{A} & \text{if } \frac{-\omega + \sqrt{\omega^2 - 4\theta\omega\sigma\chi}}{2\theta} < 1; \\
1 & \text{if } \frac{-\omega + \sqrt{\omega^2 - 4\theta\omega\sigma\chi}}{2\theta} \geq 1; \\
\frac{2\theta}{\omega} & \text{if } \frac{-\omega + \sqrt{\omega^2 - 4\theta\omega\sigma\chi}}{2\theta} \geq 1. 
\end{cases} \]
Here, we illustrate the proof for the case when the EPR is applied and the setup costs are negligible.

Proof. Let \( n \) denote the number of nodes assigned to a task. When the setup costs are negligible, the number of nodes assigned to a task is \( n^* \) such that when a task is assigned \( n^* \) nodes, the task execution time is the shortest. For example, in Fig. 5(b) \( n^* = 63 \).

This paper, for the node assignment policies, we only investigate the two extreme cases: that is, assigning \( n_{\text{min}} \) or \( n_{\text{max}} \) nodes to a task. When the setup costs are negligible, to assign \( n_{\text{max}} \) nodes means to allocate all available \( N \) nodes to a task. On the other hand, when the setup costs are significant (such that \( n^* < N \)) the strategy to assign all \( N \) nodes to a task is not a useful strategy. Instead, assigning \( n^* \) nodes can save system resources as well as minimize the task execution time.

5. Analysis of node assignment policies

While scheduling, the number of nodes assigned to a real-time divisible task could be between the minimum number \( n_{\text{min}} \) of nodes and all available \( N \) nodes. The two plots in Fig. 5 show the relationship between the task execution time \( e \) (Eq. 16) and \( n \), the number of nodes assigned, when the setup costs are different. As demonstrated in Fig. 5(a), when the setup costs are small, assigning a greater number of nodes to a task will always reduce its execution time. However, Fig. 5(b) shows that when the setup costs are significant the execution time of a task is no longer a monotonically decreasing function of the number of nodes assigned. That is, there exists an optimal number \( n^* \) such that when a task is assigned \( n^* \) nodes, the task execution time is the shortest. For example, in Fig. 5(b) \( n^* = 63 \).

In this paper, for the node assignment policies, we only investigate the two extreme cases: that is, assigning \( n_{\text{min}} \) or \( n_{\text{max}} \) nodes to a task. When the setup costs are negligible, to assign \( n_{\text{max}} \) nodes means to allocate all available \( N \) nodes to a task. On the other hand, when the setup costs are significant (such that \( n^* < N \)) the strategy to assign all \( N \) nodes to a task is not a useful strategy. Instead, assigning \( n^* \) nodes can save system resources as well as minimize the task execution time.

5.1. Further analysis on node assignment

As we will discuss later in Section 6, different node assignment policies will have their advantages in different scenarios. One of our goals is to provide guidelines for the administrators to tune the scheduling policies for the system. Thus, it is interesting to know when we should choose which node assignment policy. However, the system we study is a dynamic system. Tasks of varied sizes are submitted to the system at different times. Moreover, for each task, \( n_{\text{min}} \) also varies with the scheduled task execution start time. All these add complexities to the analysis of node assignment policies.

In this section, we first study a simplified scenario where there is only one periodic divisible task \( T \). Let \( N^* = \min(N, T_{\text{min}}) \). A new algorithm ALG-K that always assigns \( K \) nodes \((K < N^*)\) to every task is investigated. We compare it with the algorithm ALG-AN that assigns \( N^* \) nodes to each task. We will demonstrate in Section 6 that with this simple analysis we could identify some scenarios where an ALG-MN that always assigns \( n_{\text{min}} \) nodes to each task will be better than the corresponding ALG-AN when handling aperiodic divisible tasks.

In every period \( P \) we assume that a subtask (also called a job) of \( T \) with a fixed data size \( \sigma \) and a relative deadline \( D \) arrives at the cluster. We prove the following theorems.

**Theorem 5.1.** Let \( \varepsilon(\sigma, n) \) denote the execution time of a task with a data size \( \sigma \) running on \( n \) nodes. We have \( \varepsilon(\sigma, 1) < N^* \varepsilon(\sigma, N) \).

**Proof.** Here, we illustrate the proof for the case when the OPR partitioning method is applied and the setup costs are negligible. Proofs for other cases, where the EPR is applied or the setup costs are significant, are similar.

According to Eq. (8), the execution times are

\[
\varepsilon(\sigma, 1) = \sigma (\epsilon + \chi) \quad \text{and} \quad \varepsilon(\sigma, N^*) = \frac{1 - \beta}{1 - \beta^N} \sigma (\epsilon + \chi).
\]

Since \( 0 < \beta < 1 \) (as defined in Eq. (7)), dividing Eq. (23) by Eq. (24), we get

\[
\frac{\varepsilon(\sigma, 1)}{\varepsilon(\sigma, N^*)} = \frac{1 - \beta^N}{1 - \beta} = 1 + \beta + \beta^2 + \cdots + \beta^{N-1} < N^*.
\]

That is, \( \varepsilon(\sigma, 1) < N^* \varepsilon(\sigma, N) \). \( \square \)

**Corollary 5.2.** If the first job arrives at time 0, the arrival time \( A_i \) of the \( i \)th job is \((i - 1)P\). That is, \( A_i = (i - 1)P \).

**Assertion 5.3.** When an algorithm ALG-AN is applied, if a task \( T \) has a period \( P \) such that \( P \geq \varepsilon(\sigma, N) \), then the waiting time of the \( i \)th job is 0; on the other hand, if \( P < \varepsilon(\sigma, N) \), then the waiting time of the \( i \)th job is \((i - 1)P\).

According to the above Assertion, as long as \( D \geq \varepsilon(\sigma, N) \) all deadlines will be met by algorithm ALG-AN if the task period \( P \geq \varepsilon(\sigma, N) \).

**Theorem 5.4.** When an ALG-AN is applied, if \( P < \varepsilon(\sigma, N) \) and \( D \) is finite, some subtasks are doomed to miss their deadlines.

**Proof.** This follows directly from Assertion 5.3: when \( i \) is large enough, ALG-AN will miss some job’s deadline sooner or later. \( \square \)
Next, we will prove that, under certain conditions, when an ALG-K is applied, no subtasks will miss their deadlines.

**Corollary 5.5.** In any time period \([t_k, t_k+L]\), for a task \(T\) with a period \(P\) there are at most \(\lfloor \frac{P}{T} \rfloor\) job arrivals.

**Lemma 5.6.** When algorithm ALG-K is applied, if \(P \geq \frac{K\varepsilon(\sigma, K)}{N-K}\) and \(\varepsilon(\sigma, K) \geq \sigma \tau \frac{N-K}{N-K}\), no job will be delayed due to the data transmission of other jobs.

**Proof.** As discussed in Section 2, in our model, the head node sequentially sends every data chunk to its corresponding processing node via the switch. Thus, it is possible that when a job arrives at the system, it may not start even if there are enough nodes available — it has to wait for the head node to complete the data transmission for other jobs that have started. We prove this lemma by contradiction. As explained, ALG-K always assigns \(K\) nodes \((K < N')\) to every job. Assume that \(J_k\) is the first job that is delayed due to the data transmission of the jobs that have started. In this case, the data transmission time of a previous job, \(\sigma \tau\), must be greater than the period of the task; that is, \(P \leq \sigma \tau\). Combining it with one of the assumptions in the lemma, we have

\[
\frac{K\varepsilon(\sigma, K)}{N-K} \leq P < \sigma \tau.
\]

That is, \(\varepsilon(\sigma, K) < \sigma \tau \frac{N-K}{N-K}\), which contradicts the assumed condition. \(\square\)

**Lemma 5.7.** When algorithm ALG-K is applied, if \(P \geq \frac{K\varepsilon(\sigma, K)}{N-K}\) and \(\varepsilon(\sigma, K) \geq \sigma \tau \frac{N-K}{N-K}\), all jobs of the periodic task will start as soon as they arrive.

**Proof.** From Lemma 5.6, we know that no job will be delayed due to the data transmission; thus, in this case, the only reason that a job cannot start immediately upon its arrival is that the available nodes in the system are inadequate. We then prove this lemma by contradiction. As explained, ALG-K always assigns \(K\) nodes \((K < N')\) to every job. Assume that \(J_k\) is the first job that cannot start as soon as it arrives. Then at the arrival time \(A_k\) of job \(J_k\), the number of available nodes must be less than \(K\), indicating that there are another \(\left\lfloor \frac{N-K}{T} \right\rfloor\) jobs running. We order these jobs by their arrival time and get a job list \(J_1, J_2, \ldots\). Assuming that the arrival time of the first job \(J_1\) is \(A_1\), then in the time period \([A_1, A_1 + \varepsilon(\sigma, K)]\) there are \(\left\lfloor \frac{N-K}{T} \right\rfloor + 1\) job arrivals. Since job \(J_1\) is still running at time \(A_2\), we have \(A_2 < A_1 + \varepsilon(\sigma, K)\). Therefore, in the time period \([A_1, A_1 + \varepsilon(\sigma, K)]\) there are at least \(\left\lfloor \frac{N-K}{T} \right\rfloor + 1\) job arrivals. On the other hand, since \(P \geq \frac{K\varepsilon(\sigma, K)}{N-K}\), according to Corollary 5.5 the number \(m\) of job arrivals in \([A_1, A_1 + \varepsilon(\sigma, K)]\) should be at most \(\left\lfloor \frac{N-K}{T} \right\rfloor\) because

\[
m = \left\lfloor \frac{\varepsilon(\sigma, K)}{\varepsilon(\sigma, K)} \right\rfloor \leq \left\lfloor \frac{\varepsilon(\sigma, K)}{\varepsilon(\sigma, K)} \right\rfloor = \left\lfloor \frac{N-K}{K} \right\rfloor = \left\lfloor \frac{N'-K}{K} \right\rfloor - 1 \leq \left\lfloor \frac{N'-K}{K} \right\rfloor.
\]

We find a contradiction. \(\square\)

**Theorem 5.8.** When algorithm ALG-K is applied, if \(P \geq \frac{K\varepsilon(\sigma, K)}{N-K}\), \(\varepsilon(\sigma, K) \geq \sigma \tau \frac{N-K}{N-K}\), and \(D \geq \varepsilon(\sigma, K)\), then all jobs of the periodic task will meet their deadlines.

**Proof.** According to Lemma 5.7, all subtasks will start as soon as they arrive. When the deadline \(D \geq \varepsilon(\sigma, K)\), all subtasks will meet their deadlines. \(\square\)

From Theorems 5.4 and 5.8, we conclude that if \(P \in \left(\frac{K\varepsilon(\sigma, K)}{N-K}, \varepsilon(\sigma, N')\right)\), \(\varepsilon(\sigma, K) \geq \sigma \tau \frac{N-K}{N-K}\), and \(D \geq \varepsilon(\sigma, K)\) for some \(K \in [1, N']\), an ALG-K will perform better than the corresponding ALG-AN.

Later in Section 6 we will discuss how we use these theorems to guide the choice of node assignment policies for aperiodic divisible tasks.

### 6. Performance evaluation

In previous sections, we have proposed and analyzed various real-time cluster-based scheduling algorithms for divisible loads. In this section, their performance relative to each other and to changes of configuration parameters are experimentally evaluated.

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 component, a task dispatcher, and a logging component. The real-time scheduler component is implemented following our algorithm framework proposed in Section 3.4, which can be configured to simulate different scheduling algorithms with varied policies on task execution order, workload partitioning and node assignment.

For each simulation, five parameters, \(N, \tau, \chi, \theta_{\text{tm}}\), and \(\theta_{\text{sp}}\), are specified for a cluster. In this paper, to evaluate the algorithms’ performance in processing different streams of tasks, we generate synthetic workloads with parameters varying in wide ranges. To generate task \(T_i = (A_i, \sigma_i, D_i)\), similar to the work by Lee et al. [23], we assume that the interarrival times follow an exponential distribution with a specified mean of \(1/\lambda\) and task data sizes \(\sigma_i\) are
normally distributed with a specified mean of \( \text{Aug} \ \sigma \) and a standard deviation equal to the mean. Task relative deadlines are assumed to be uniformly distributed in the \( \left[ \frac{\text{Avg} \ \sigma}{2}, \frac{2 \cdot \text{Avg} \ \sigma}{2} \right] \) range, where \( \text{Avg} \ D \) is the mean relative deadline. To specify \( \text{Avg} \ D \), a new term DC Ratio is introduced. It is defined as the ratio of the mean deadline to the mean execution time (cost), that is \( \frac{\text{Avg} \ D}{\text{Avg} \ \sigma} \). In this way, by DC Ratio, task relative deadlines are specified relating to the average task execution time. In addition, a task relative deadline \( D \) is chosen to be larger than its execution time \( \frac{\text{Avg} \ D}{\text{Avg} \ \sigma} \).

Similar to the work by Lee et al. [23], we define another metric SystemLoad to represent how loaded a cluster is:

\[
\text{SystemLoad} = \frac{\frac{\text{Avg} \ \sigma}{1}\lambda}{N},
\]

where \( \frac{\text{Avg} \ \sigma}{1}\lambda \) is the execution time of an average size task running on a processing node and \( \lambda / N \) is the average task arrival rate per node. Sometimes, we specify SystemLoad for a simulation instead of the average interarrival time \( 1/\lambda \). Configuring \( (N, \tau, \chi, \theta_m, \theta_p, \text{SystemLoad}, \text{Avg} \ \sigma, \text{DC Ratio}) \) is equivalent to specifying \( (N, \tau, \chi, \theta_m, \theta_p, 1/\lambda, \text{Avg} \ \sigma, \text{DC Ratio}) \), because

\[
\frac{\text{Avg} \ \sigma}{1}\lambda = \text{SystemLoad} \times N.
\]

To evaluate the real-time performance, we use two metrics: the Task Reject Ratio and System Utilization. The 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 all figures in this paper, a point on a curve corresponds to the average performance value of ten simulations. In the ten runs, the same parameters \( (N, \tau, \chi, \theta_m, \theta_p, 1/\lambda, \text{Avg} \ \sigma, \text{DC Ratio}) \) are specified but different random numbers are generated for task arrival times \( A_i \), data sizes \( \sigma_i \), and deadlines \( D_i \). For each simulation, the total simulation time is 10,000,000 time units, which is sufficiently long.

We have identified three important scheduling decisions, Task Partitioning, Node Assignment, and Scheduling Policy, in designing real-time, cluster-based scheduling algorithms for divisible loads (see Section 3). In the next three subsections, we evaluate the effects of these decisions, compare the algorithms proposed in Section 3, and respectively investigate the scenarios where each of these three decisions matters.

### 6.1. OPR vs. EPR partitioning

We first evaluate the performance of the following real-time scheduling algorithms with respect to the two proposed partitioning rules (OPR and EPR): EDF-OPR-MN vs. EDF-EPR-MN, EDF-OPR-AN vs. EDF-EPR-AN, FIFO-OPR-MN vs. FIFO-EPR-MN, FIFO-OPR-AN vs. FIFO-EPR-AN, and MWF-OPR-MN vs. MWF-EPR-MN. We only present the comparisons of EDF-OPR-MN vs. EDF-EPR-MN and EDF-OPR-AN vs. EDF-EPR-AN here. The performance results for the other pairs are similar.

**Simulation modeling.** For our basic simulation model we chose the following parameters: number of processing nodes in the cluster \( N = 256 \); unit data transmission time \( \tau = 1 \); unit data processing time \( \chi = 1000 \); transmission setup cost \( \theta_m = 500 \); processing setup cost \( \theta_p = 500 \); SystemLoad changes in the \( [0.1, 0.2, \ldots, 1.0] \) range; the average data size \( \text{Avg} \ \sigma = 1000 \); and the ratio of the average deadline to the average execution time \( \text{DC Ratio} = 2 \). Our simulation has a three-fold objective. First, we want to justify our hypothesis that it is advantageous to apply DLT in real-time cluster-based scheduling. Second, we study the effects of DC Ratio, and third, we want to investigate the effects of the processing speed.

#### 6.1.1. Merits of DLT for cluster scheduling

To study the merits of DLT we employ our basic simulation model without any change. Fig. 6 shows Task Reject Ratio and System Utilization of the four algorithms: EDF-OPR-MN, EDF-EPR-MN, EDF-OPR-AN, and EDF-EPR-AN. Observe that EDF-OPR-MN always leads to a lower Task Reject Ratio and a higher System Utilization than EDF-EPR-MN. Similarly, EDF-OPR-AN always performs better than EDF-EPR-AN. These simulation results confirm our hypothesis that it is advantageous to apply DLT in real-time cluster-based scheduling algorithms. The reason is, compared to the partitioning heuristic EPR, the DLT-based OPR provides an optimal task partitioning, which leads to minimum task execution times. As a result, with an OPR scheduling algorithm (i.e., EDF-OPR-MN or EDF-OPR-AN), the cluster can satisfy a larger number of task deadlines and be better utilized.

We carried out the same type of simulations by changing the following cluster or workload parameters one at a time: cluster size \( N \) and average data size \( \text{Avg} \ \sigma \). The results are similar to those in Fig. 6, where algorithms with OPR partitioning always perform better than algorithms with EPR partitioning.

#### 6.1.2. Effects of DC Ratio

To study the effects of the DC Ratio, we use the same configuration as the basic simulation model except that we vary the DC Ratio over \( [2, 4, 6, 10, 20, 50, 100] \). For the sake of readability, Fig. 7 only shows the performance of EDF-OPR-AN and EDF-EPR-AN with \( \text{DC Ratio} = 2, 10, \) and 100. Corresponding to different combinations of algorithm and DC Ratio, six curves are produced. Again, Fig. 7 shows that the algorithm with OPR partitioning performs better. In addition, we can see that when SystemLoad is low (i.e., when SystemLoad < 0.8), the performance of EDF-EPR-AN becomes closer to that of EDF-OPR-AN as DC Ratio increases. This is because the higher the DC Ratio, the looser the task deadlines are. Consequently, when SystemLoad is low and cluster resources are plenty, the worse execution times caused by a non-optimal partitioning rule, like the EPR, will have less impact on the algorithm’s performance.

#### 6.1.3. Effects of processing speed

To study the effects of the processing speed, we vary \( \chi \) over the \( [100, 500, 1000, 5000, 10000] \) range. The larger the \( \chi \), the slower the computation. Fig. 8 shows the results of EDF-OPR-MN and EDF-EPR-MN with \( \chi = 100, 1000, \) and 10,000 respectively. We observe that the OPR partitioning algorithm EDF-OPR-MN still outperforms the EPR partitioning algorithm EDF-EPR-MN. However, as the processing speed decreases, i.e., \( \chi \) increases, the differences between the two algorithms become less significant. In particular, when the computation is extremely slow \( (\chi = 10000) \), the curves for the two algorithms are almost the same, indicating non-differentiable Task Reject Ratios and System Utilization. To demonstrate this point, let us assume that \( \chi \) is so large that the ratio of \( \tau \) to \( \chi \) is approaching 0. As a result, \( \beta \) from Eq. (7) will approach 1, causing the data fractions allocated to processing nodes \( \alpha_1, \alpha_2, \ldots, \alpha_n \), to all be close to \( \frac{1}{n} \) for the OPR. Therefore, the OPR and the EPR will perform the same in this case.

**Summary.** From the aforementioned intensive experiments, we have the following conclusions. (a) No matter what the system parameters are, the algorithms with DLT-based partitioning (OPR) always perform better than those with the equal-sized partitioning heuristic (EPR). This demonstrates that it is beneficial to apply DLT (divisible load theory) in real-time cluster-based scheduling. (b) When SystemLoad is low, the difference between the OPR and the EPR becomes smaller as DC Ratio (i.e., deadline) increases. (c) As \( \chi \) increases, that is, as node processing speed decreases, the difference between the OPR and the EPR becomes negligible.
6.2. \( n^* \) vs. \( n^{\text{min}} \) node assignment

In this subsection, we compare and analyze the real-time scheduling algorithms with different node assignment methods. We investigate the performance difference in algorithms assigning all \( N \) or \( n^* \) nodes to every task (ALG-AN) vs. those assigning the minimum number \( n^{\text{min}} \) of nodes needed to meet a task deadline (ALG-MN). The relative performance of EDF-OPR-MN vs. EDF-OPR-AN is systematically studied. It is noteworthy that in contrast to the results by Lee et al. [23] comparing MWF(-MN) and FIXED(-AN) algorithms, our initial data (see Fig. 6) seem to indicate that EDF-OPR-AN outperforms EDF-OPR-MN most of the time.

6.2.1. Effects of transmission cost

Fig. 9(a) shows the relative performance of the two algorithms, i.e., Task Reject Ratio (TRR) of EDF-OPR-MN — Task Reject Ratio (TRR) of EDF-OPR-AN. In this simulation, we gradually increase the transmission cost \( \tau \). As we can see, when \( \tau \) is small EDF-OPR-MN leads to a bigger Task Reject Ratio than EDF-OPR-AN and as \( \tau \) increases EDF-OPR-MN begins to have a smaller Task Reject Ratio than EDF-OPR-AN. This indicates that the relative performance of EDF-OPR-MN vs. EDF-OPR-AN improves as \( \tau \) gets larger.

In Section 5, we have discussed the rationale behind the two different node assignment strategies: an algorithm of type ALG-AN tries to finish the current task as soon as possible by assigning more processing nodes to a task, while an algorithm of type ALG-MN tries to conserve resources for new tasks. For an ALG-AN, the problem is it causes higher parallel execution overheads than the ALG-MN counterpart, e.g., EDF-OPR-AN leads to higher overheads than EDF-OPR-MN. As shown in Fig. 2(a), the node idle time due to data transmission is one type of parallel execution overhead. For the cluster model investigated (see Section 2), the higher the transmission cost \( \tau \) the greater the overhead. That explains why in the aforementioned simulation we observe that, as \( \tau \) increases, the performance of EDF-OPR-AN is affected more than that of EDF-OPR-MN and EDF-OPR-MN begins to perform better than EDF-OPR-AN.

The results shown in Figs. 6 and 9(a) contradict the conclusion drawn by Lee et al. [23] that the \( n^* \) node assignment strategy (ALG-MN) performs better than the maximum node assignment.
strategy (ALG-AN). As we can see in Fig. 9(a), there are scenarios where ALG-MN performs better than ALG-AN, while in the other scenarios the reverse is true.

6.2.2. Effects of DC Ratio

In this subsection, we study the effects of changing the deadlines, where we vary the DC Ratio from 2 to 10. By increasing the DC Ratio, we have longer relative deadlines compared to the mean execution time. For an ALG-MN, a longer deadline leads to a smaller $n_{\text{min}}$ of nodes allocated to a task, thus smaller parallel execution overhead, while for an ALG-AN, its node assignment and resulting overhead will not be affected by deadlines, since a task is always assigned $\min(N, n^*)$ number of nodes. Therefore, we believe, as DC Ratio increases and ALG-MN’s overhead decreases, ALG-MN’s performance relative to that of ALG-AN is going to improve. Fig. 9(b) validates our hypothesis, where we observe that by increasing DC Ratio from 2 to 10, the Task Reject Ratio difference of EDF-OPR-MN and EDF-OPR-AN gets smaller.

6.2.3. Leveraging the analysis results

In Section 5.1 we theoretically characterize scenarios where algorithms of type ALG-K outperform their counterparts of type ALG-AN. This subsection demonstrates that the insight gained from the analysis can be leveraged to derive the scenarios where ALG-MN is guaranteed to outperform its ALG-AN counterpart. For an ($N = 64, \tau = 1, \chi = 100, \theta_{\text{sm}} = 0, \theta_{\text{sp}} = 0$) cluster that runs aperiodic tasks of size $\sigma = 200$, we derive the task interarrival time $(1/\lambda)$ ranges where EDF-OPR-K, $K \in \{1, 2, \ldots, 8\}$, algorithms outperform the EDF-OPR-AN algorithm (Table 1). We can see that the common subrange of the 8 ranges is $[366, 425]$. Thus, if task interarrival times fall into that range and the task relative deadlines are long enough that EDF-OPR-MN always computes $n_{\text{min}} \leq 8$, EDF-OPR-MN will perform better than EDF-OPR-AN.

We conducted two simulations, one comparing EDF-OPR-MN vs. EDF-OPR-AN and the other comparing FIFO-OPR-MN vs. FIFO-OPR-AN. A scenario as described above is created. Simulation results show that, for such a configuration, EDF-OPR-MN and FIFO-OPR-MN both have 0 Task Reject Ratios while EDF-OPR-AN and FIFO-OPR-AN’s Task Reject Ratios are 0.0523 and 0.0564, respectively. We thus verify that under the derived conditions ALG-MN indeed performs better than its ALG-AN counterpart.

Summary. From the aforementioned intensive experiments, we have the following conclusions. (a) Both ALG-MN and ALG-AN have their own advantages. One outperforms the other under certain system configurations and workload scenarios. (b) As the transmission cost $\tau$ increases, the relative performance of ALG-MN vs. ALG-AN improves. (c) As DC Ratio increases and the task deadlines become less tight, ALG-MN’s performance gets better relative to that of ALG-AN.

6.3. FIFO, EDF vs. MWF scheduling policies

In this subsection, we examine different execution order policies and compare algorithms FIFO-OPR-MN, EDF-OPR-MN vs. MWF-OPR-MN.

Recall that the MWF (Maximum Workload derivative First) algorithm proposed by Lee et al. [23] executes the task with the highest workload derivative ($\delta w_i$) first and thus reduces the total workload (cost) of all scheduled tasks. In their paper [23] MWF is compared with EDF and it is shown that MWF performs better than EDF. Moreover, the authors claim that MWF is likely to be the best choice for on-line scheduling of divisible tasks.

We conducted intensive simulations and a systematic study of the three execution order strategies. Our data cast some doubts on the conclusion drawn by Lee et al. [23] that the MWF algorithm is the best choice. Our hypothesis is that MWF performs well when task parallel execution overhead (workload) is significant compared to pure task computation time. To test our hypothesis, a group of simulations was designed to study how changing parallel overhead affects the performance of scheduling algorithms. In the 20 simulations, we gradually change the data transmission cost ($\tau$) from 1 to 20, while keeping the data processing cost ($\chi$) constant. Since the bigger the $\tau$ the higher the parallel execution overhead, for the 20 simulations with $\tau$ changing from 1 to 20 the task overhead increases. According to our theory, MWF should perform better than EDF and FIFO when $\tau$ increases.

Fig. 10 shows the results for simulations where $\tau = 1, 10$ and 20, respectively. As observed, when $\tau$ is small, the Task Reject Ratio curve of EDF-OPR-MN lies below that of MWF-OPR-MN, indicating that the EDF execution order performs better. As $\tau$ increases, the relative performance of the two algorithms begins to change. When $\tau$ increases to 20, MWF-OPR-MN outperforms EDF-OPR-MN, leading to smaller Task Reject Ratios for most System Load conditions. These data match our analysis and verify our hypothesis that MWF performs better than EDF and FIFO as the workload parallelization overhead increases.

Interestingly, for all 20 simulations with $\tau$ changing from 1 to 20, EDF-OPR-MN always leads to smaller Task Reject Ratios than FIFO-OPR-MN. Another quite interesting phenomenon is that, among the three algorithms, MWF-OPR-MN always results in the worst System Utilizations. MWF policy tries to schedule tasks with bigger workload derivatives $\delta w_i$ first (see Eq. (1) for $\delta w_i$’s calculation). The larger the task size $\sigma$, the bigger $\delta w_i$ tends to be. Thus, this scheduling policy tries to schedule large tasks first. However, inserting those large tasks before small tasks often causes deadline violations of small tasks. As a result, with MWF policy, large tasks usually cannot pass the schedulability test and are likely to be rejected, which explains why MWF-OPR-MN leads to
Table 1
1/\( \lambda \) ranges where EDF-OPR-K outperforms EDF-OPR-AN.

<table>
<thead>
<tr>
<th>( K )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>([316,425))</td>
<td>([318,425))</td>
<td>([335,425))</td>
<td>([321,425))</td>
<td>([350,425))</td>
<td>([357,425))</td>
<td>([366,425))</td>
<td>([327,425))</td>
</tr>
</tbody>
</table>

Summary. From the discussion above, we conclude that: (a) The best choice of execution order policy depends on individual system and workload conditions; (b) our results seem to show at most of the time algorithms using EDF policy perform better than algorithms using FIFO policy; (c) when the communication cost (\( \tau \)) is small, algorithms using the MWF policy do not have advantages, while as \( \tau \) increases, MWF algorithms begin to perform better than their EDF and FIFO counterparts; and (d) MWF algorithms tend to reject large tasks and thus lead to smaller system utilisations.

7. Related work

The development of high-performance computing environments has gained considerable momentum. By linking a large number of computers together, a cluster provides a cost-effective facility for solving complex problems. A resource management system (RMS), which provides real-time guarantees or QoS, is central to cluster performance.

Utility-driven cluster computing [40,35,22,28] has been well studied to improve the value of utility delivered to users. Proposed cluster RMSs [12,3] have addressed the scheduling of both sequential and parallel loads. The goal of those schemes is similar to ours:

---

**Fig. 10.** FIFO, EDF vs. MWF: When \( \tau = 1, 10, 20 \).
to harness the power of resources based on user objectives. They differ from ours, however, in the task and system models assumed.

The scheduling models investigated for distributed or multiprocessor systems most often (e.g., [32,31,12,19,15,22]) assume periodic or aperiodic sequential jobs that must be allocated to a single resource and executed by their deadlines. In recent years, researchers have begun to investigate real-time scheduling of parallel applications on a cluster [41,30,18,24]. However, most of these studies assume the existence of some form of task graph to describe the communication and precedence relations between computational units called subtasks (i.e., nodes in the task graph). Netto and Buyya [28] consider the scheduling of parallel bag-of-tasks applications, where each application is formed of a bag of independent sequential tasks that need to be completed by a deadline. Because bag-of-tasks applications are not arbitrarily divisible, they are different from the divisible loads investigated in this paper.

The most closely related work [23] to ours is scheduling algorithms for “scalable real-time tasks” running on a multiprocessor system. In that paper, like divisible loads, it is assumed that a task can be executed on more than one processor, and as more processors are allocated, its pure computation time decreases monotonically. The paper notes that the decision on the number of processors allocated to tasks is an important factor in the design of parallel scheduling algorithms. However, the simulations described in the paper are limited. Their conclusions on comparing their proposed MWF (Maximum Workload derivative First) schemes with the EDF and FIXED algorithms [27,7] hold true only in certain scenarios (see Section 6.3 for a detailed explanation). The work on scheduling “moldable jobs” [8,14,20,34,36] is also related, but only He et al. [20] have considered QoS support.

Our work differs significantly from previous work in real-time as well as cluster computing in both the task model assumed and in the comprehensiveness of our study. We have published our preliminary work in a conference paper [24]. This journal paper significantly extends that work [24]: here we propose and evaluate new algorithms that can handle setup costs of divisible loads. In addition, detailed analysis of node assignment policies and complete theorem proofs are included in this paper.

8. Conclusion

In this paper, we address the problem of providing deterministic QoS to arbitrarily divisible applications executing in cluster environments. For real-time scheduling of divisible loads, three important design decisions need to be made: (1) workload partitioning, (2) node assignment, and (3) task execution order. By systematizing a study of real-time divisible load scheduling algorithms, we have made many interesting discoveries. In particular, we found that, for real-time cluster-based divisible load scheduling, it is beneficial to apply DLT (divisible load theory) to guide workload partitioning; a proper node assignment is determined by system and workload conditions and there is a need for adaptive node assignment strategies; and MWF execution order leads to good real-time performance when task parallel execution overheads are high.

References

Xuan Lin  is a Ph.D. student in the Department of Computer Science and Engineering, University of Nebraska–Lincoln. His research interests include grid computing, real-time scheduling and divisible load theory. Xuan Lin received his M.S. and B.S. degrees in Computer Science from Southeast University, Nanjing, China, in 2001 and 1998, respectively.

Anwar Mamat is a Ph.D. student in the Department of Computer Science and Engineering, University of Nebraska–Lincoln. He received his B.E. and M.S. degrees from Chang’an University, Xi’an, China, in 1999 and 2002, respectively. His research interests are grid and cluster computing, high-performance computing, and real-time scheduling.

Ying Lu received her B.S. degree in Computer Science in 1996 from the Southwest Jiaotong University, Chengdu, China, and her M.C.S. and Ph.D. degrees in Computer Science in 2001 and 2005, respectively, from the University of Virginia. Since August 2005, she has been at the University of Nebraska–Lincoln, where she is an Assistant Professor in the Department of Computer Science and Engineering. Her research interests include real-time systems, autonomic computing, cluster and grid computing, and web systems. She has done significant work on the feedback control of computing systems.

Jitender Deogun received his B.S. (1966) with Honors from Punjab University, Chandigarh, India, and his M.S. (1974), and Ph.D. (1979) degrees from University of Illinois, Urbana, IL. He joined the University of Nebraska in 1981 as an Assistant Professor of Computer Science. He was promoted to Associate Professor in 1984 and Full Professor in 1988. His research interests include high-performance optical switch architectures, routing and wavelength assignment, real-time scheduling, network coding, and structural and algorithmic graph theory.

Steve Goddard is a Professor and Chair in the Department of Computer Science and Engineering at the University of Nebraska–Lincoln. He received his B.A. degree in Computer Science and Mathematics from the University of Minnesota (1985). He received his M.S. and Ph.D. degrees in Computer Science from the University of North Carolina at Chapel Hill (1995 and 1998, respectively). His research interests are embedded, real-time and distributed systems with an emphasis in real-time, rate-based scheduling theory.