

Real-Time Divisible Load Scheduling for Cluster Computing

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

Abstract

Cluster Computing has emerged as a new paradigm for solving large-scale problems. To enhance QoS and provide performance guarantees in cluster computing environments, various workload models and real-time scheduling algorithms have been investigated. The *divisible load model*, propagated by divisible load theory, models computations that can be arbitrarily divided into independent pieces and provides a good approximation of many real-world applications. However, researchers have not yet investigated the problem of providing performance guarantees to divisible load applications. Two contributions are made in this paper: (1) divisible load theory is extended to compute the minimum number of processors required to meet an application's deadline; and (2) the first cluster-based, real-time scheduling algorithm designed specifically for arbitrarily divisible loads is presented and evaluated.

1 Introduction

The dawn of the information age has changed how we solve important problems. Emerging computation and data intensive applications cannot be solved by a single stand-alone machine. This has led to the emergence of *cluster computing* as a new paradigm for computing. Cluster computing harnesses the power of hundreds and thousands of machines to facilitate the computation of large and complex problems in many application domains. However, as the size of a cluster increases, so does the complexity of resource management and maintenance. Thus, innovations in automated performance control and resource management are crucial for continued evolution of cluster computing. On one hand, system administrators prefer a system that is easy to manage. On the other hand, end-users expect high performance from the cluster, such as receiving computational results before specified deadlines.

The challenge, however, in applying real-time schedul-

ing theory to cluster computing is that computational loads submitted to clusters are structured in various ways. Some, called sequential jobs, are difficult to compute concurrently whereas others are comprised of tasks that can be executed in parallel. Parallel jobs can be further categorized based on the divisibility property of their computational loads. Modularly divisible loads can be subdivided a priori into a certain number of subtasks; these loads are often described with a task (or processing) graph. Arbitrarily divisible loads can be partitioned into any number of load fractions, and are quite common in high energy and particle physics. For example, the the CMS (Compact Muon Solenoid) [9] and ATLAS (AToroidal LHC Apparatus) [5] projects, which are associated with the Large Hadron Collider (LHC) at CERN (European Laboratory for Particle Physics), execute cluster-based applications with arbitrarily divisible loads.

The cluster and real-time computing research communities have thoroughly explored the problem of providing QoS or real-time guarantees for sequential jobs and modularly divisible jobs in distributed systems. Similarly, significant progress has been made in *divisible load theory* [26]. However, despite the increasing importance of arbitrarily divisible applications [21], to the best of our knowledge, the real-time scheduling of arbitrarily divisible loads has not been addressed before.

This creates a problem for cluster-based research computing facilities such as the U.S. CMS Tier-2 sites that are building high-end clusters for CMS applications [23], which may execute for days or even weeks. (The CMS project will not be fully operational until 2007. Thus, the actual work load generated by this world-wide experiment can only be simulated at this time.) One of the management goals of 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 each job [23]. Existing real-time cluster-based scheduling algorithms assume the existence of a task graph for all applications, while divisible load the-

ory attempts to minimize schedule length with no regard for the actual deadline.

Two contributions are made in this paper: (1) divisible load theory is extended to compute the minimum number of processors required to meet an application deadline; and (2) the first cluster-based, real-time scheduling algorithm designed specifically for arbitrarily divisible loads is presented and evaluated. Henceforth, the term “divisible” means “arbitrarily divisible” unless specified otherwise.

The remainder of this paper is organized as follows. Section 2 presents related work, and Section 3 describes the task and system models assumed. Extensions to divisible load theory to support real-time scheduling are presented in Section 4, while Section 5 presents the scheduling algorithm. Section 6 evaluates the performance of the algorithm. Section 7 presents our conclusions.

2 Related Work

Development of commodity-based clusters and Grid computing have recently gained considerable momentum. By linking a large number of computers together, a cluster provides cost-effective power for solving complex problems. In a large-scale Grid, the resource management system (RMS) is central to its operation. In order to serve end-users in a timely fashion, it is essential for the underlying cluster RMS to provide performance guarantees or QoS.

Research has been carried out in utility-driven cluster computing [27, 22] to improve the value of utility delivered to the users. Proposed cluster RMSs [7, 3] have addressed the scheduling of both sequential and parallel programs. The goal of those schemes is similar to ours—to harness the power of resources based on user objectives.

The real-time computing community, has made significant progress in scheduling of periodic and/or aperiodic tasks with deadlines in distributed or multiprocessor systems. The models investigated most often, e.g., in [20, 19, 13, 1, 17, 12], assume periodic or aperiodic sequential jobs that must be allocated to a single resource and executed by its deadline. With the evolution of cluster computing, researchers have begun to investigate real-time scheduling of parallel applications on a cluster, e.g., [29, 18, 11, 2, 4]. However, [29, 18, 11, 2, 4] all assume the existence of some form of task graph to describe communication and precedence relations between computational units called subtasks (i.e., nodes in the task graph).

The most closely related work is [15], wherein the authors propose scheduling algorithms for “scalable real-time tasks” on multiprocessor systems. It is assumed in their model that tasks can be executed on more than

one processors and that task computation times decrease monotonically as more processors are allocated. We show that this assumption is not true when communication costs are considered. Moreover, unlike their work, which assumes the task execution time function is known a priori, this paper applies divisible load theory to dynamically compute the task execution time functions.

Our work differs significantly from other work in real-time as well as cluster computing in both the task model assumed and in the computational resources available. As described in Section 3, we assume a workload in which each aperiodic task is arbitrarily divisible into independent subtasks (i.e., no precedence relations or inter-subtask communication) that can be executed in parallel on a cluster of computers scheduled by a head node.

Divisible load theory [6, 21, 26] provides an in-depth study of distribution strategies for arbitrarily divisible loads in multiprocessor/multicomputer systems subject to system constraints like link speed, processor speed and interconnection topology. The goal of divisible load theory is to exploit parallelism in computational data so that the workload can be partitioned and assigned to several processors such that execution completes in the shortest possible time [6]. The application of divisible load theory is widespread [21]. An example related to our work is its application to [28, 14] and implementation in [25] Grid computing. Complimentary to other work, our paper applies divisible load theory to the design of a real-time scheduling algorithm for cluster computing; specifically, divisible load theory is applied to the scheduling of applications, such as CMS [9] and ATLAS [5], that execute on a large cluster.

3 Task and System Models

Task Model. We investigate real-time scheduling of arbitrarily divisible tasks that arrive aperiodically and execute non-preemptively (once subtasks are allocated to processors). In the real-time aperiodic task model each aperiodic task T_i typically consists of a single invocation specified by the tuple (A_i, C_i, D_i) , where $A_i \geq 0$ is the arrival time of the task, $C_i > 0$ is its computational requirement, and $D_i > 0$ is the relative deadline. The absolute deadline of the task is given by $A_i + D_i$. The computational requirement C_i is usually considered to be the worst case execution time of the task. The aperiodic task model adopted here, however, uses the data size σ_i to represent the computational requirement. That is, a divisible task $T_i = (A_i, \sigma_i, D_i)$ is a single invocation, where A_i is the arrival time of the task, σ_i is the total data size of the task, and D_i is the relative deadline. Task execution time is dynamically determined using σ_i and allocated resources—processing nodes and bandwidth—by lever-

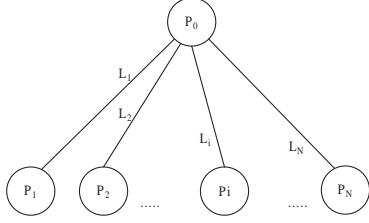


Figure 1: System Topology.

aging the modelling power of divisible load theory [26], as explained in Section 4.

System Model. A cluster consists of head node, denoted P_0 , and N processing nodes, denoted by P_1, \dots, P_N . 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, and divide and distribute the workload to processing nodes. A star network topology (see Figure 1) is used to represent the communication requirements of the cluster. Since tasks and subtasks are independent, there is no need for processing nodes to communicate with each other.

In this work, a homogenous cluster and sequential transmission of the workload is assumed. That is, (1) all processing nodes have the same computational power; (2) all links from the head node to the processing nodes have the same bandwidth; and (3) the head node does not begin to distribute the workload to node P_{i+1} until it has completed its workload transmission to node P_i . According to divisible load theory, linear models are used to represent processing speeds and transmission times [26]. In the simplest scenario, the computation time of a load σ 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 σ is calculated by a cost function $Cm(\sigma) = \sigma C_{ms}$, where C_{ms} is the time for a link to transmit a unit workload. Divisible load theory also provides models for heterogeneous networks [26], which will be used in the future to extend this work to heterogeneous clusters.

4 Task Partition and Execution Time Analysis

Executing a divisible load in a cluster entails two decisions—*allocating processing nodes to the task* and *partitioning the task load among the allocated processing nodes*. Divisible load theory states that optimal execution time is obtained for a divisible load if all processing

nodes allocated to the task complete their computation at the same time instant [26]. This is called the *Optimal Partitioning Rule* (or simply, OPR). Development of our cluster scheduling algorithm is guided by the OPR.

In divisible load theory, normally all n nodes of a cluster are allocated to a task. Then, following the OPR, the task load is partitioned such that all nodes finish processing at the same time. In contrast to this approach, we first compute the minimum number of processing nodes needed to meet the task’s deadline, and then partition the task following the OPR (using at least the minimum number of nodes required to meet the deadline). The execution time of a task is then trivially computed as the difference between its completion and start times. The following notations, partially adopted from [26], will be used in these computations.

- $T = (A, \sigma, D)$: A divisible task, where A = arrival time, σ = data size, and D = relative deadline
- $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_n)$: Data distribution vector, where $0 < \alpha_j < 1$ and $\sum_{j=1}^n \alpha_j = 1$
- α_j : Data fraction allocated to the j^{th} processing node
- C_{ps} : Processing time for a unit workload
- C_{ms} : Time for transporting a unit workload
- ST : The setup time (cost) for the head node to initialize communication on a link
- SC : The setup time (cost) for a processing node to initialize a computation

We analyze the task execution time under two different models [26]. In the first model (Section 4.1), we assume there are no setup costs for initializing data communication and computation. In the second model (Section 4.2), we consider the communication and computation setup costs.

4.1 Analysis without Setup Cost

Assuming no set up cost, we now compute a task’s execution time and the minimum number of nodes needed to meet its deadline on a homogeneous system. Based on our system model (Section 3) we have the following cost functions.

Processing time on j^{th} node: $C_p(\alpha_j \sigma) = \alpha_j \sigma C_{ps}$;
 Transport time on j^{th} link: $C_m(\alpha_j \sigma) = \alpha_j \sigma C_{ms}$.

The OPR leads to the timing diagram in Figure 2 when n nodes are allocated to a task load. Let \mathcal{E} denote *Task*

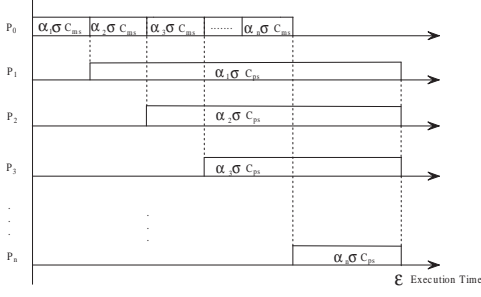


Figure 2: Timing diagram of system without setup cost.

Execution Time and \mathcal{C} denote *Task Completion Time*. By analyzing the diagram, we have

$$\mathcal{E} = \alpha_1 \sigma C_{ms} + \alpha_1 \sigma C_{ps} \quad (4.1)$$

$$= (\alpha_1 + \alpha_2) \sigma C_{ms} + \alpha_2 \sigma C_{ps} \quad (4.2)$$

$$= (\alpha_1 + \alpha_2 + \alpha_3) \sigma C_{ms} + \alpha_3 \sigma C_{ps} \quad (4.3)$$

...

$$= (\alpha_1 + \alpha_2 + \alpha_3 + \dots + \alpha_n) \sigma C_{ms} + \alpha_n \sigma C_{ps} \quad (4.4)$$

From (4.1) and (4.2), we have

$$\alpha_1 = \alpha_2 \frac{\sigma C_{ms} + \sigma C_{ps}}{\sigma C_{ps}} = \frac{\alpha_2}{\beta}, \quad \text{where}$$

$$\beta = \frac{\sigma C_{ps}}{\sigma C_{ms} + \sigma C_{ps}} = \frac{C_{ps}}{C_{ms} + C_{ps}}. \quad (4.5)$$

It follows that $\alpha_2 = \beta \alpha_1$. Similarly, from (4.2) and (4.3), we have $\alpha_3 = \beta \alpha_2$, and therefore, $\alpha_3 = \beta^2 \alpha_1$. This leads to a general formula: $\alpha_j = \beta^{j-1} \alpha_1$. Since α_j is the data fraction distributed to j^{th} processing node, we have $\sum_{j=1}^n \alpha_j = 1$, and substituting α_j with $\beta^{j-1} \alpha_1$ in this equation, we obtain

$$\alpha_1 + \beta \alpha_1 + \beta^2 \alpha_1 + \dots + \beta^{n-1} \alpha_1 = 1.$$

Solving this equation, we get $\alpha_1 = \frac{1-\beta}{1-\beta^n}$. Thus, the execution time, \mathcal{E} , for the task is

$$\mathcal{E} = \alpha_1 \sigma (C_{ms} + C_{ps})$$

$$= \frac{1-\beta}{1-\beta^n} \sigma (C_{ms} + C_{ps}).$$

Assuming that task $T = (A, \sigma, D)$ has start time s , then $\mathcal{C} = s + \mathcal{E} \leq A + D$, because the task must satisfy its deadline. It follows that,

$$s + \frac{1-\beta}{1-\beta^n} \sigma (C_{ms} + C_{ps}) \leq A + D. \quad \text{Thus}$$

$$\frac{1-\beta}{1-\beta^n} \sigma (C_{ms} + C_{ps}) \leq A + D - s. \quad (4.6)$$

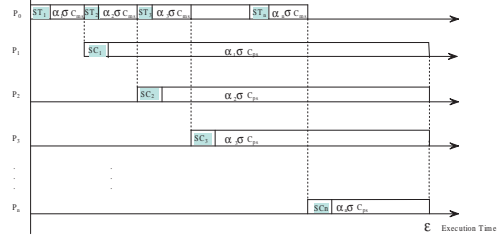


Figure 3: Timing diagram of system with setup cost.

Since $\beta = \frac{C_{ps}}{C_{ms} + C_{ps}} < 1$, $1 - \beta^n > 0$. Multiplying both sides of Eq. (4.6) by $(1 - \beta^n)$, we get

$$(1 - \beta) \sigma (C_{ms} + C_{ps}) \leq (1 - \beta^n) (A + D - s).$$

If $(A + D - s) \leq 0$, the task will miss its deadline no matter how we schedule it at time s . Therefore, assuming $(A + D - s) > 0$ and dividing both sides by $A + D - s$, we have

$$(1 - \beta^n) \geq \frac{(1 - \beta) \sigma (C_{ms} + C_{ps})}{A + D - s}. \quad \text{Thus,}$$

$$\beta^n \leq 1 - \frac{(1 - \beta) \sigma (C_{ms} + C_{ps})}{A + D - s}$$

$$= 1 - \frac{(1 - \frac{C_{ps}}{C_{ms} + C_{ps}}) \sigma (C_{ms} + C_{ps})}{A + D - s}$$

$$= 1 - \frac{(\frac{C_{ms}}{C_{ms} + C_{ps}}) \sigma (C_{ms} + C_{ps})}{A + D - s}$$

$$= 1 - \frac{\sigma C_{ms}}{A + D - s}$$

$$= \gamma$$

where $\gamma = 1 - \frac{\sigma C_{ms}}{A + D - s}$, and it follows that $n \geq \frac{\ln \gamma}{\ln \beta}$. Since n is an integer, $n \geq \lceil \frac{\ln \gamma}{\ln \beta} \rceil$. Therefore, the minimum number of processing nodes that the task needs to complete before its deadline at time s is $n^{\min} = \lceil \frac{\ln \gamma}{\ln \beta} \rceil$ where γ is defined above and β in (4.5).

4.2 Analysis with Setup Cost

The setup cost of communication and computation cannot be ignored in practice. The setup cost of communication comes from physical network latencies, network protocol overhead, or middleware overhead. In the TeraGrid project [24], the network speed can be up to 40Gbit/Sec with latency around 100ms, which means around 1/3 of the time required to send 1GB of data is due to latency. [8] shows that the setup cost for computation can be up to 25 seconds in practice, which is also not neglectable for some small tasks.

We now consider the communication and computation setup cost to derive the task execution time and the minimum number of processing nodes needed for the task to meet its deadline. The processing time on the j^{th} node is $C_p(\alpha_j\sigma) = SC + \alpha_j\sigma C_{ps}$, and the transmission time on the j^{th} link is $C_m(\alpha_j\sigma) = ST + \alpha_j\sigma C_{ms}$. The timing diagram with setup cost (SC) is shown in Figure 3. As before, based an analysis of the timing diagram, we have

$$\begin{aligned}\mathcal{E} &= (ST + \alpha_1\sigma C_{ms}) + (SC + \alpha_1\sigma C_{ps}) \quad (4.7) \\ &= 2ST + (\alpha_1 + \alpha_2)\sigma C_{ms} + \\ &\quad (SC + \alpha_2\sigma C_{ps}) \quad (4.8)\end{aligned}$$

$$\begin{aligned}&= 3ST + (\alpha_1 + \alpha_2 + \alpha_3)\sigma C_{ms} + \\ &\quad (SC + \alpha_3\sigma C_{ps}) \quad (4.9)\end{aligned}$$

$$\begin{aligned}\dots \\ &= (n-1)ST + \quad (4.10) \\ &\quad (\alpha_1 + \alpha_2 + \alpha_3 + \dots + \alpha_n)\sigma C_{ms} + \\ &\quad (SC + \alpha_n\sigma C_{ps})\end{aligned}$$

From (4.7) and (4.8), we have $\alpha_2 = \alpha_1\beta - \phi$, where β is defined in (4.5) and $\phi = \frac{ST}{\sigma(C_{ms} + C_{ps})}$. Similarly, from (4.8) and (4.9), we get $\alpha_3 = \alpha_2\beta - \phi$, and therefore $\alpha_3 = \alpha_1\beta^2 - \beta\phi - \phi$, leading to the general formula

$$\begin{aligned}\alpha_j &= \alpha_1\beta^{j-1} - \sum_{k=0}^{j-2}\beta^k\phi. \quad \text{Thus,} \\ \alpha_j &= \alpha_1\beta^{j-1} - \frac{1 - \beta^{j-1}}{1 - \beta}\phi.\end{aligned}$$

Now, substituting α_j with $(\alpha_1\beta^{j-1} - \frac{1 - \beta^{j-1}}{1 - \beta}\phi)$ in the equation $\sum_{j=1}^n \alpha_j = 1$, we get

$$\begin{aligned}\sum_{j=1}^n (\alpha_1\beta^{j-1} - \frac{1 - \beta^{j-1}}{1 - \beta}\phi) &= 1 \\ \implies \sum_{j=0}^{n-1} (\alpha_1\beta^j - \frac{1 - \beta^j}{1 - \beta}\phi) &= 1.\end{aligned}$$

A solution to the above equation leads to

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

Assuming $\mathcal{B} = \frac{1 - \beta}{1 - \beta^n} + \frac{n\phi}{1 - \beta^n} - \frac{\phi}{1 - \beta}$, it follows that $\mathcal{E} = ST + SC + \sigma(C_{ms} + C_{ps})\mathcal{B}$ and as before if task $T = (A, \sigma, D)$ has start time s , then, $\mathcal{E} \leq A + D - s$. It follows that,

$$ST + SC + \sigma(C_{ms} + C_{ps})\mathcal{B} \leq A + D - s. \quad (4.11)$$

Thus, the smallest integer greater than or equal to n that satisfies the above constraint is the minimum number of processing nodes that need to be assigned to task T at time s to satisfy its deadline. This constraint can be solved numerically.

Note that the model without setup cost (Section 4.1) is a special case of this model, where $ST = SC = 0$ and accordingly, $\phi = \frac{ST}{\sigma(C_{ms} + C_{ps})} = 0$. Therefore, we can reduce constraint (4.11) to constraint (4.6), $\sigma(C_{ms} + C_{ps})\frac{1 - \beta}{1 - \beta^n} \leq A + D - s$, which was derived for the model without setup cost.

5 Dynamic Scheduling of Divisible Loads

In this section, we present an algorithm for scheduling real-time arbitrarily divisible loads, consisting of aperiodic tasks dispatched dynamically. The problem of dynamic scheduling on multiprocessor systems, without a priori knowledge of task arrival times is NP-complete [?, 10]. This motivates our heuristic approach to solve the problem of *dynamic scheduling of divisible loads*.

Like typical dynamic scheduling [10, 19, 16], when new tasks arrive, our scheduler dynamically determines the feasibility of scheduling the new tasks without compromising the guarantees for the previously admitted tasks. This feasibility analysis is done before a task is admitted to the cluster. A feasible schedule is generated if the deadlines of all tasks in the cluster can be satisfied. Tasks will be dispatched according to the feasible schedule developed. If no feasible schedule is found, the task will be rejected, meaning in the cluster environment that the system administrator will negotiate with the client about the feasible deadline for the task.

Before describing the details of our algorithm, we introduce the following notations (some of them are adopted from [15]).

$n_i^{min}(t)$: the minimum number of processing nodes needed to finish computation of task T_i , dispatched at time t , before its deadline.

$W_i(n) = n * \mathcal{E}$: cost of task T_i when n processing nodes are assigned to it. (see Figure 2).

$DC_i = W_i(n_i^{min} + 1) - W_i(n_i^{min})$: the derivative of $W_i(n)$ with respect to n evaluated at its current n_i^{min} .

The proposed scheduling algorithm, called *Maximum Cost Derivative First* (MCDF), allocates the minimum number of processing nodes to a task that satisfies its deadline; and a task with high cost derivative is favored to start earlier, just as [15] does.

The motivation for the heuristic is to minimize the total cost of all scheduled tasks. It is assumed that the smaller the total cost of the current tasks, the more likely that the newly arrived tasks will meet their deadlines [15]. Next,

we prove why following these rules minimizes the total cost of the divisible tasks in the cluster environment.

Contrary to the scalable task model assumed in [15], we prove that for divisible load model with setup cost (Section 4.2), as the number of processing nodes allocated to a task increases, its computation time *does not* decrease monotonically. However, for the divisible load model without setup cost (Section 4.1) the assertion does hold.

Theorem 5.1 *For a divisible load model without setup cost, when the number of processing nodes assigned to a task increases, its execution time decreases monotonically.*

Proof: See [?].

Theorem 5.2 *For a divisible load model with setup cost, when the number of processing nodes assigned to a task increases, its execution time does not decrease monotonically.*

Proof: See [?].

We believe these theorems have important implications for the design of scheduling algorithms for divisible loads in a cluster computing environment. We design our algorithm accordingly.

Next, we prove that the cost $W_i(n)$ of computation increases monotonically as the number of nodes allocated to a divisible task T_i increases.

Theorem 5.3 *When the number of processing nodes assigned to a task increases, the total cost $W_i(n)$ increases monotonically.*

Proof: See [?].

To summarize the discussion and theorems above, we have several rules to follow for development of the proposed heuristic: 1) to minimize the total cost, the number of processing nodes assigned to each task is set at its current minimum, i.e., $n_i^{min}(t)$; 2) schedule tasks in order of decreasing cost derivative, i.e., always schedule the task T_i with the highest cost derivative DC_i first.

Data Structures and Algorithm. We now present the data structures used in the algorithm, followed by the algorithm.

- *NILList* $\langle j, t_j \rangle$: *Node-Information-List*. The list stores the information about computation nodes, where j denotes the index of the computation node and t_j denotes the time when the node becomes idle.
- *AdmittedTasksList* $\langle i, t_{arrival_i}, D_i, \sigma_i, s_i, e_i, n_i^{min} \rangle$. The list stores the tasks that have been admitted but not been dispatched, where i denotes the index of the task, $t_{arrival_i}$ is the arrival time of the task, D_i is the

relative deadline of the task, σ_i is the workload of the task, s_i will be the starting time of the task, e_i will be the completion time of the task, and n_i^{min} will be the minimum number of computation nodes the task needs at time s_i to complete before its deadline.

- *NewTasksList* $\langle i, t_{arrival_i}, D_i, \sigma_i \rangle$. The list stores the tasks which just arrive at the system, where i denotes the index of the task, $t_{arrival_i}$ is the arrival time of the task, D_i is the relative deadline of the task, σ_i is the workload of the task.
- *AvailableNodesList* $\langle t_k, AN_k \rangle$. The list is a list of number of available nodes along with the time, where t_k is the time and AN_k is the number of available nodes at time t_k . This list can be generated based on the information of *NILList*.
- *UnScheduledTasksList* $\langle i, t_{arrival_i}, D_i, \sigma_i, s_i, e_i, n_i^{min}, DC_i \rangle$. This list stores the tasks that have not been scheduled, where there is an additional term DC_i representing the derivative cost of T_i at n_i^{min} . The structure of this list is the same as *AdmittedTasksList*.
- *TempSTList*: *Temporarily-Scheduled-Tasks-List*. The data structure of this list is the same as *AdmittedTasksList*. It stores the tasks that have been temporarily scheduled at the Schedulability-Test stage. If the Schedulability-Test is passed, meaning that the admitted tasks and the new task are all schedulable before their deadlines, the new task will be admitted and the temporary schedule will be accepted, that is, we will overwrite the *AdmittedTasksList* with the *TempSTList*, which includes the new scheduling information.

The pseudo code of our algorithm, called Maximum Cost Derivative First (MCDF) is as follows.

```

1. void MCDF()
2.   while true
3.     if AdmittedTasksList != ∅
4.       for each  $T_i$  in AdmittedTasksList
5.         if starting time  $s_i$  == current_time
6.           dispatch Task  $T_i$  to  $n_i^{min}$  nodes
7.           remove  $T_i$  from AdmittedTasksList
8.           update NILList
9.       end for
10.    if NewTasksList != ∅
11.      for each  $T_i$  in NewTasksList
12.        if Schedulability_Test( $T_i$ ) == true
13.          accept  $T_i$ 
14.          /* accept the new schedule */
15.          AdmittedTasksList ← TempSTList
16.        else
17.          reject  $T_i$ .
```

```

17.   end for
18. end while
19. end MCDF()

1. boolean Schedulability_Test( $T_i$ )
2.   UnScheduledTasksList  $\leftarrow$  AdmittedTasksList +  $T_i$ 
3.   generate AvailableNodesList /* from NIList */
4.   TempSTList  $\leftarrow$   $\emptyset$ 
5.   /* index for AvailableNodesList <  $t_k, AN_k$  >*/
6.    $k \leftarrow 1$ 
7.   while UnScheduledTasksList  $\neq$   $\emptyset$ 
8.     for each  $T_i$  in UnScheduledTasksList
9.       calculate  $n_i^{min}(t_k)$  and  $DC_i$ 
10.      /* N: total processing node number */
11.      if  $n_i^{min}(t_k) > N$ 
12.        return false /*not schedulabe*/
13.      end for
14.      /* by nonincreasing order of  $DC_i$  */
15.      order UnScheduledTasksList
16.      /* from the head to the tail of the list */
17.      for each  $T_i$  in UnScheduledTasksList
18.        if  $n_i^{min}(t_k) \leq AN_k$ 
19.          /* set scheduled starting time */
20.           $s_i \leftarrow t_k$ 
21.          /* set expected completion time */
22.           $e_i \leftarrow \mathcal{E}(\sigma_i, n_i^{min}(t_k)) + t_k$ 
23.          if  $e_i > t_{arrival_i} + D_i$ 
24.            return false /* deadline misses */
25.          remove  $T_i$  from UnScheduledTasksList
26.          insert  $T_i$  into TempSTList
27.          update AvailableNodesList
28.          /* if no more idle nodes at time  $t_k$  */
29.          if  $AN_k == 0$ 
30.            break
31.          end for
32.           $k++$ ;
33.        end while
34.        /* all tasks in the cluster are schedulable */
35.      return true
36.    end Schedulability_Test()

```

Note that this scheduling algorithm may cause a fragmentation where processing nodes are idle. In our future work, we plan to reduce processing idle times by leverage multi-round divisible load scheduling [6].

6 Performance Evaluation

We use a discrete simulator to model the system and evaluate the proposed scheduling algorithm with respect to the metric *Task Miss/Rejected Ratio* (or simply *Task Miss Ratio*). The Task Miss/Rejected Ratio is of the number of tasks that miss their deadline to the total number of tasks that arrived at the system for algorithms without admission control and for the algorithms with admission control, this is the ratio of the number of tasks that are rejected by the scheduler to the total number of task that arrived at the system. Thus, our algorithm focuses on minimizing the task miss ratio.

6.1 Simulation Setup

The system load, L , is defined as the sum of minimum execution time (using all nodes) of all tasks divided by the total simulation time. The data sizes of tasks are assumed to be normally distributed with a mean 100 and a standard deviation equal to the mean. The deadlines of tasks are chosen to be larger than the minimum computation time and are assumed to be uniformly distributed between the minimum and maximum computation time. The number of computational nodes in the system is assumed to be ten.

We assume a Poisson task arrival process, with average interval time of Poisson Distribution is defined as the average minimum execution time of tasks divided by system load. At each arrival point, the number of tasks arriving is randomly chosen number between one and ten, both inclusive. The simulation time is set as 1,000,000 time units which is considered to be sufficiently long. The simulation is run ten times and the mean value is computed.

6.2 Comparative Evaluation without Setup Cost

We compare our algorithm with six popular algorithms. The six algorithms belong to two group. First group is the FIFO (First In First Out). The survey [?] shows that although the current prominent commercial cluster management software suites like Moab/Maui, LoadLeveler, LSF, PBS, SGE, OSCAR, are packaged with a several schedulers, but the default setting is FIFO. The second group is EDF (Early Deadline First), which is known to be the optimal algorithm on single processor systems when tasks are not preemptive. Algorithms in both group are further divided in to three types: *without admission control, using all nodes for every task*, and *using minimum nodes for every task*. Thus, these algorithms are FIFOANNA (FIFO using All Nodes and No Admission control), FIFOAN (FIFO using All Nodes),

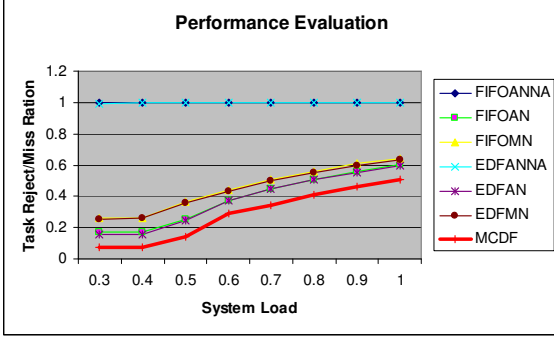


Figure 4: Performance Evaluation-1 (without startup cost)

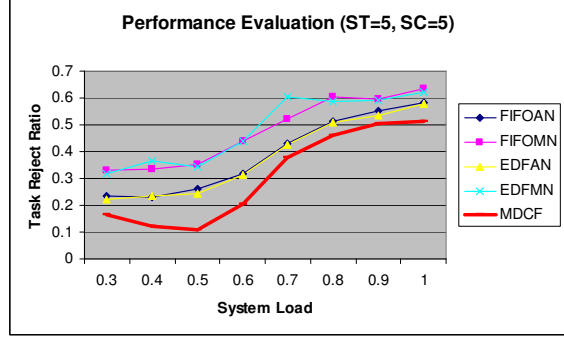


Figure 6:

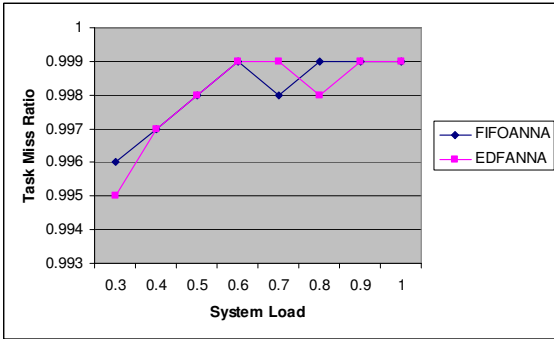


Figure 5: Performance Evaluation-2 (without startup cost)

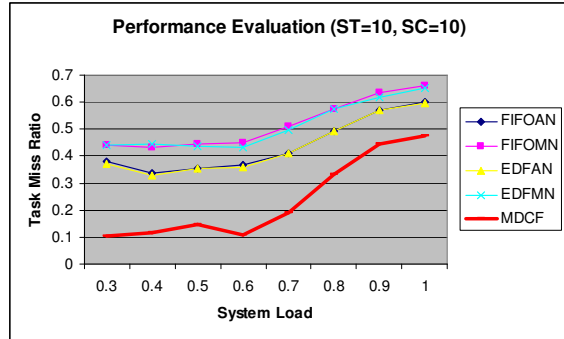


Figure 7:

FIFOMN (FIFO using Minimum number of Nodes), EDFANNA (EDF using All Nodes and No Admission control), EDFAN (EDF using All Nodes) and EDFMN (EDF using Minimum number of Nodes).

Figures 6.2 and 6.2 compare the performance of the proposed algorithm MCDF to the six algorithms described above. The algorithm MCDF, performs much better as compared to all six algorithms. We observe that algorithms without admission control miss the deadlines for more that 99% of tasks. This is because the delays do propagate. Among the six algorithms the performance of FIFOAN and EDFAN is close to MCDF, but MCDF still exhibit better performance than either with a margin of about 10% decrement of Task Reject Ratio.

6.3 Comparative Evaluation with Set-up Cost

Figures Figures 6.3, 6.3, 6.3 and 6.3 show the comparative performance of our algorithm with respect to the six algorithms when setup cost is considered. Since the algorithms without admission control do not really perform well, we do not consider them here. The simula-

tion setups are the same as we described in section 6.1 except that the values of ST and SC are set from 5 to 20. From these graph, we can see that MDCF, our algorithm, still has the best performance. Furthermore, it can be observed that as setup cost increases, the gain in performance of MDCF over other algorithms increases. Under our simulation setup, MDCF exhibits much more stability than the four algorithms (FIFOAN, FIFOMN, EDFAN, EDFMN). Moreover, as the of setup cost increases, the Task Miss Ratio increases for the four algorithms, while the Task Miss Ratio of MDCF remains relatively unchanged.

6.4 Impact of C_{ms} and C_{ps}

In this section we study the impact of changing the ratio of C_{ms} to C_{ps} , that is, the ratio of communication cost to the computation cost. These two parameters are most significant parameters and thus sensitivity of our algorithm to change in their ratio is of significance.

We first study the impact of the ratio on the task completion time. We fix the data size of the task to be 100, the number of computational nodes to 10, and C_{ps} to 10. We simulate the effect of changing the ratio of C_{ms} to C_{ps} from 0.1 to 10.

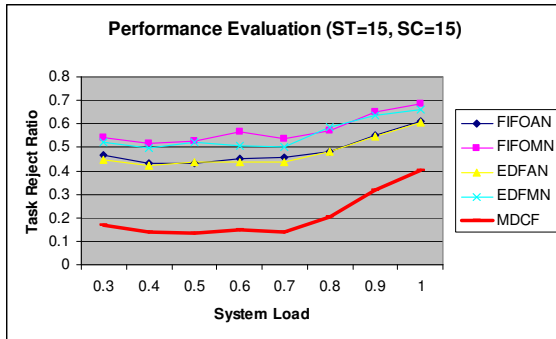


Figure 8:

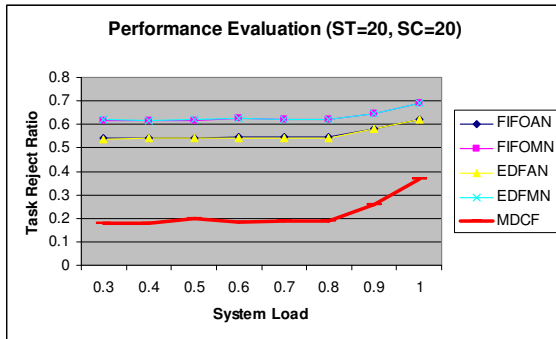


Figure 9:

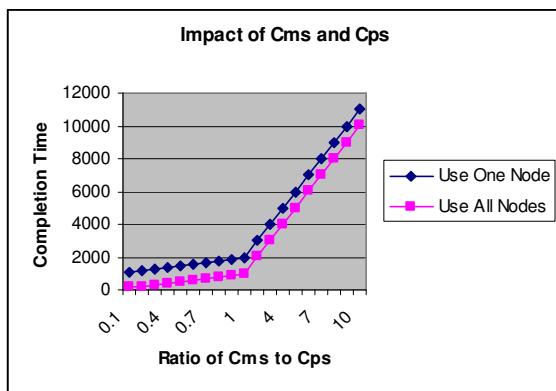


Figure 10:

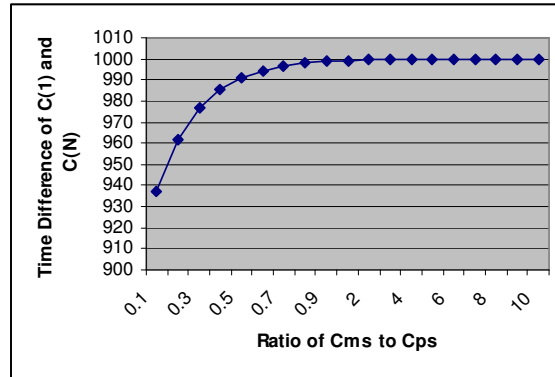


Figure 11:

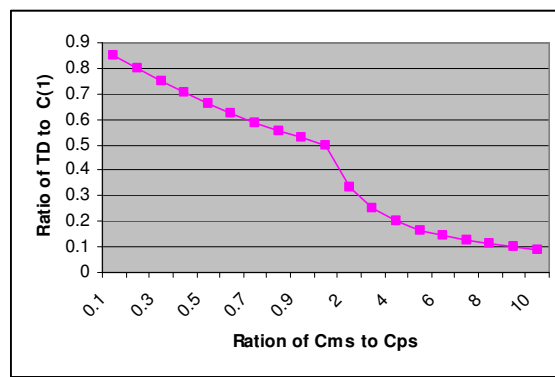


Figure 12:

Let $C(1)$ denote the completion time of a task when only one node is assigned to it, and $C(N)$ the completion time of the task when all nodes in the cluster are assigned. Again, let $TD = C(1) - C(N)$ denote the difference in the two completion times.

From Figure 6.4, we observe that when the ratio of C_{ms} to C_{ps} increases, as expected the completion time of the task also increases. Analysis of Figures 6.4, 6.4 and 6.4 lead to several interesting observations. From all three figures, we conclude that although TD increases as the the ratio of C_{ms} to C_{ps} increases but the the increment levels off before the ratio reaches 2.0. An analysis of Figure 6.4, depicts the gain that can be obtained in completion time of a task by load distribution among the nodes of the cluster. However, the gain levels off as ratio of C_{ms} to C_{ps} approaches 0.9. Figure 6.4, again shows the gain we can obtain from load distribution. The figure shows the ratio of TD to $C(1)$, which indicates the extent of benefit from parallelization of the task, decreases sharply as the ratio of C_{ms} to C_{ps} increase.

We can infer from above discussion that as the transport cost increases with respect to computation cost the completion time is adversely affected.

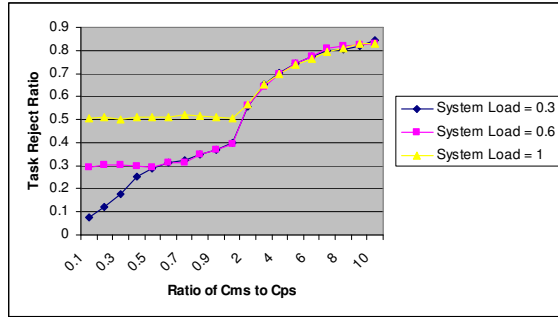


Figure 13:

Finally, we study the impact of the ratio of C_{ms} to C_{ps} on the performance of our algorithm. From Figure 6.4, we can observe that when the ratio of C_{ms} to C_{ps} small, the Task Reject Ratio of our algorithm is very sensitive to the system load. However, the sensitivity of our algorithm system load decreases as the ratio of C_{ms} to C_{ps} increase. Moreover, the algorithm loses all sensitivity to system load as the ratio of C_{ms} to C_{ps} increases beyond 3.0.

References

- [1] T. F. Abdelzaher and V. Sharma. A synthetic utilization bound for aperiodic tasks with resource requirements. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS 2003)*, pages 141–150, Porto, Portugal, July 2003.
- [2] A. Amin, R. Ammar, and A. E. Dessouly. Scheduling real time parallel structure on cluster computing with possible processor failures. In *Proceedings of the Ninth IEEE International Symposium on Computers and Communications (ISCC 2004)*, pages 62–67, July 2004.
- [3] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren. An opportunity cost approach for job assignment in a scalable computing cluster. *IEEE Transactions on Parallel and Distributed Systems*, 11(7):760+, 2000.
- [4] R. A. Ammar and A. Alhamdan. Scheduling real time parallel structure on cluster computing. In *Proceedings of the Seventh IEEE International Symposium on Computers and Communications (ISCC 2002)*, pages 69–74, Taormina, Italy, July 2002.
- [5] ATLAS (AToroidal LHC Apparatus) Experiment, CERN (European Laboratory for Particle Physics). Atlas web page. <http://atlas.ch/>.
- [6] V. Bharadwaj, T. G. Robertazzi, and D. Ghose. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.
- [7] B. N. Chun and D. E. Culler. Market-based proportional resource sharing for clusters. Technical Report UCB/CSD-00-1092, EECS Department, University of California, Berkeley, 2000.
- [8] G. Chun, H. Dail, H. Casanova, and A. Snaveley. Benchmark probes for grid assessment. In *IPDPS*, 2004.
- [9] Compact Muon Solenoid (CMS) Experiment for the Large Hadron Collider at CERN (European Laboratory for Particle Physics). Cms web page. <http://cmsinfo.cern.ch/Welcome.html/>.
- [10] M. L. Dertouzos and A. K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Trans. Softw. Eng.*, 15(12):1497–1506, 1989.
- [11] M. Eltayeb, A. Dogan, and F. Özgüner. A data scheduling algorithm for autonomous distributed real-time applications in grid computing. In *Proceedings of the 33rd International Conference on Parallel Processing (ICPP 2004)*, pages 388–395, Montreal, Quebec, Canada, August 2004.
- [12] S. Funk and S. Baruah. Task assignment on uniform heterogeneous multiprocessors. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS 2005)*, pages 219–226, July 2005.
- [13] D. Isovich and G. Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *Proc. of the 21st IEEE Real-Time Systems Symposium*, Orlando, Florida, USA, November 2000.
- [14] S. Kim and J. B. Weissman. A genetic algorithm based approach for scheduling decomposable data grid applications. In *Proceedings of the International Conference on Parallel Processing (ICPP04)*, pages 406–413, Montreal, Quebec, Canada, August 2004.
- [15] W. Y. Lee, S. J. Hong, and J. Kim. On-line scheduling of scalable real-time tasks on multiprocessor systems. *J. Parallel Distrib. Comput.*, 63(12):1315–1324, 2003.
- [16] G. Manimaran and C. S. R. Murthy. An efficient dynamic scheduling algorithm for multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):312–319, 1998.
- [17] P. Pop, P. Eles, Z. Peng, and V. Izosimov. Schedulability-driven partitioning and mapping for multi-cluster real-time systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 2004)*, pages 91–100, July 2004.
- [18] X. Qin and H. Jiang. Dynamic, reliability-driven scheduling of parallel real-time jobs in heterogeneous systems. In *Proceedings of the 30th International Conference on Parallel Processing (ICPP 2001)*, pages 113–122, Valencia, Spain, September 2001.
- [19] K. Ramamritham, J. A. Stankovic, and P. fei Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, April 1990.
- [20] K. Ramamritham, J. A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Trans. Comput.*, 38(8):1110–1123, 1989.
- [21] T. G. Robertazzi. Ten reasons to use divisible load theory. *Computer*, 36(5):63–68, 2003.
- [22] J. Sherwani, N. Ali, N. Lotia, Z. Hayat, and R. Buyya. Libra: a computational economy-based job scheduling system for clusters. *Software: Practice and Experience*, 34(6):573–590, 2004.
- [23] D. Swanson. Personal communication. Director of UNL Research Computing Facility (RCF) and UNL CMS Tier-2 Site, August 2005.
- [24] TERAGRID.
- [25] K. van der Raadt, Y. Yang, and H. Casanova. Practical divisible load scheduling on grid platforms with apst-dv. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, CA, USA, April 2005.

- [26] B. Veeravalli, D. Ghose, and T. G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–17, 2003.
- [27] C. S. Yeo and R. Buyya. A taxonomy of market-based resource management systems for utility-driven cluster computing. *Software: Practice and Experience*, accepted in Sept. 2005.
- [28] D. Yu and T. G. Robertazzi. Divisible load scheduling for grid computing. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, Los Angeles, CA, USA, November 2003.
- [29] L. Zhang. Scheduling algorithm for real-time applications in grid environment. In *Proceedings of the 2002 IEEE International Conference on Systems, Man and Cybernetics*, October 2002.