A Strategyproof Mechanism for Scheduling Divisible Loads in Linear Networks

Thomas E. Carroll and Daniel Grosu

Wayne State University
Dept. of Computer Science
Detroit, MI 48202, USA
{tec, dgrosu}@cs.wayne.edu

Abstract

In this paper we augment DLT (Divisible Load Theory) with incentives such that it is beneficial for processors to report their true processing capacity and compute their assignments at full capacity. We propose a strategyproof mechanism with verification for scheduling divisible loads in linear networks with boundary load origination. The mechanism provides incentives to processors for reporting deviants. The deviants are penalized which abates their willingness to deviate in the first place. We prove that the mechanism is strategyproof and satisfies the voluntary participation condition.

1. Introduction

Scheduling is one of the most studied topics in distributed systems. This paper considers the problem of scheduling divisible loads which is characterized by large data sets where every element within the set requires an identical type of processing. The set can be partitioned into any number of fractions where each fraction requires scheduling.

Divisible Load Theory (DLT) studies the scheduling of divisible loads in distributed systems considering different network architectures [6]. DLT assumes that the processors are obedient, *i.e.*, they do not "cheat" or perform any action that is not explicitly prescribed by the algorithm. This assumption is not valid in the real world systems where the processors are owned and operated by autonomous, self-interested organizations that have no *a priori* motivation for cooperation and they are tempted to manipulate the algorithms in hope of increased benefits. Considering this type of environment, the processors should be properly modeled as *strategic* agents. New protocols for DLT must account for this self-interested behavior. Mechanism design theory takes into account the selfishness of the participants and provides a framework for designing such protocols. *Mechanism design theory* is a field of economics that has recently garnered interest in computer science. It addresses *incentive compatibility* where *rational* agents, which are characterized as self-interested and utility-maximizing, are provided incentives which induce a behavior that maximizes the social welfare. An agent is parametrized by private values. A *strategyproof mechanism* results in a participant maximizing its utility if and only if it truthfully reports its private parameters and follows the specified algorithm. Each agent in a general mechanism has a *valuation function* that quantifies the agent's benefit. The mechanism awards *payments* to the participants in order to motivate them to report their true valuation. An agent's objective is to maximize its utility which is the sum of the valuation and payment. In the context of divisible load scheduling, we have several resource providers that offer processor time. We assume that each resource is characterized by its job processing rate. A load allocation mechanism assigns load to each resource. The allocation mechanism is strategyproof if and only if a resource owner maximizes her utility by reporting the true processing rate to the mechanism. Furthermore, the utility is independent of the values reported by the other participants.

In our previous work [9, 14], we showed how DLT can be augmented with incentives. We designed strategyproof mechanisms for scheduling divisible loads in *bus and tree networks* comprising strategic processors. The mechanisms provide incentives to the processors to participate and to report their true processing rate. The agents maximize their welfare by truthfully reporting their values to the mechanism and executing their assignments at full capacity.

Our contributions. In this paper, we augment DLT with incentives for scheduling divisible loads in *linear networks* comprising strategic processors. We propose a strate-

1-4244-0910-1/07/$20.00 ©2007 IEEE.

gyproof mechanism with verification for scheduling divisible loads in linear networks. The mechanism solves the scheduling problem in linear networks with boundary load origination. The mechanism is an example of *autonomous node mechanism* [17], where the agents (*i.e.*, the processors) have control over *both* the inputs to the algorithm and the algorithm itself. The self-interested processors will implement an algorithm different from what is prescribed if it is beneficial to do so. To cope with this scenario, processors are provided incentives to report deviants. The mechanism penalizes the deviants, which abates their willingness to deviate.

Related work. Recently, the divisible load scheduling problem was studied extensively resulting in a cohesive theory called Divisible Load Theory (DLT) [2, 3, 6, 7, 19, 21]. New results and open research problems in DLT are presented in [3]. A wide range of applications used DLT algorithms to schedule loads [4, 5, 8, 10, 16]. All these works assumed that the participants in the load scheduling algorithms are obedient. Recently, several researchers considered the mechanism design theory to solve several computational problems that involve self-interested participants. These problems include task scheduling [20], routing [11] and multicast transmission [12]. In their seminal paper, Nisan and Ronen [18] considered for the first time the mechanism design problem in a computational setting. They proposed and studied a VCG (Vickrey-Clarke-Groves) type mechanism for the shortest path in graphs where edges belong to self interested agents. They also provided a mechanism for solving the problem of scheduling tasks on unrelated machines. A general framework for designing strategyproof mechanisms for one parameter agent was proposed by Archer and Tardos [1]. They developed a general method to design strategyproof mechanisms for optimization problems that have general objective functions and restricted form for valuations. The results and the challenges of designing distributed mechanisms are surveyed in [13]. Mitchell and Teague [17] extended the distributed mechanism in [12] devising a new model where the agents themselves implement the mechanism, thus allowing them to deviate from the algorithm. Grosu and Chronopoulos [15] proposed a strategyproof mechanism that solves the static load balancing problem in distributed systems. Strategyproof mechanisms with verification combining incentives and DLT were proposed by Grosu and Carroll [9, 14].

Organization. The paper is structured as follows. In Sect. 2 we describe the divisible load scheduling problem in the context of linear networks. In Sect. 3 we discuss the mechanism design foundations. In Sect. 4 we present our proposed mechanism. In Sect. 5 we prove the properties of our new mechanism. In Sect. 6 we draw conclusions and present future directions.

2. Divisible Load Scheduling

We consider a distributed system comprising $m + 1$ processors connected in a *linear network*. Processor P_i ($i = 0, \ldots, m$) is characterized by w_i, which is the time it takes to process a unit load. The processor is assigned α_i units of load and it takes time $\alpha_i w_i$ to compute its assignment, which corresponds to a linear cost model. If the entire load to be scheduled is one unit, then $0 \leq \alpha_i \leq 1$. We assume that the processors have front-ends that permit simultaneous communication and processing. Further, a sender may communicate with only one recipient at any instant, *i.e.*, we assume the one-port model. A processor can begin computing as soon as it has received its entire assignment. The load originates at the *root*, which we designate to be processor P_0. Processor P_{j-1} ($j = 1, \ldots, m$) transmits $D_j = 1 - \sum_{k=0}^{j-1} \alpha_k$ units of load to its successor P_j in time $D_j z_j$, where z_j is the time it takes to communicate a unit load from P_{j-1} to P_j over link ℓ_j. We denote by $\boldsymbol{\alpha} = (\alpha_0, \ldots, \alpha_m)$ the vector of load allocations. Processor P_i finishes its assignment in time $T_i(\boldsymbol{\alpha})$, which is the total time to receive, transmit, and compute.

There are two types of linear networks differentiated by the location of the root processor. The linear network with *boundary load origination* has the root processor at one of the network extremes, *i.e.*, processor P_0 is a terminal processor. In the case of a linear network with *interior load origination* the root is an interior processor with two directly-connected neighbors. In this paper we consider a linear network with boundary load origination.

We use the following assumptions in characterizing the models: (i) The communication startup time is negligible; (ii) The time for passing messages in the network is negligible when compared to the time taken for communication and processing of computational loads; (iii) The time taken for returning the result of the load processing back to the root is small.

Figure 1 illustrates a $(m + 1)$-processor linear network with boundary load origination. In Figure 2, we present a Gantt chart depicting the execution time of the system. The communication time is represented above the time axis and the computation time is represented below the time axis. We denote by $\mathbf{P} = (P_0, \ldots, P_m)$ the processor set composing the network. From the diagram, it is easily observed that

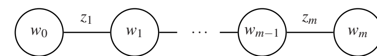

Figure 1. An $(m+1)$-processor linear network with boundary load origination.

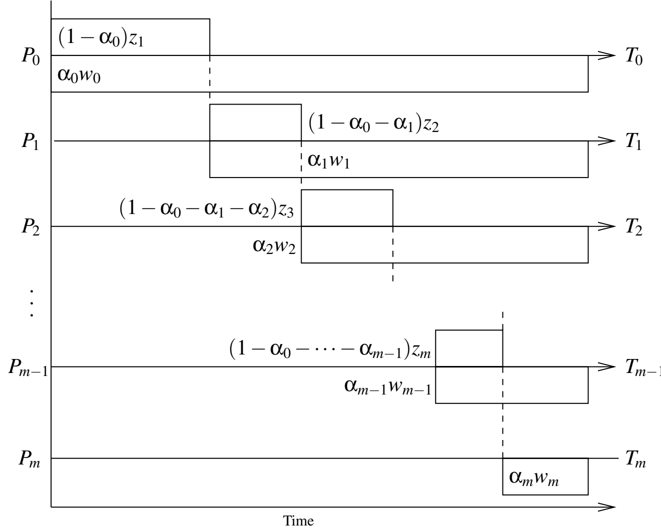

Figure 2. Execution on a $(m+1)$-processor linear network with the load originating at the boundary.

the finishing time $T_i(\boldsymbol{\alpha})$ is

$$T_0(\boldsymbol{\alpha}) = \alpha_0 w_0 \tag{2.1}$$

and for $j = 1, \ldots, m$

$$T_j(\boldsymbol{\alpha}) = \begin{cases} 0 & \text{if } \alpha_j = 0, \\ \sum_{k=1}^{j} \left(1 - \sum_{\ell=0}^{k-1} \alpha_\ell\right) z_k + \alpha_j w_j & \text{if } \alpha_j > 0. \end{cases} \tag{2.2}$$

We associate a scheduling problem with the system described above. We call this problem LINEAR BOUNDARY-LINEAR. The two words before the hyphen identify the network type and the word following the hyphen identifies the cost model. The goal of this problem is to solve for the optimal load allocation $\boldsymbol{\alpha}$ which minimizes the total execution time $T(\boldsymbol{\alpha}) = \max(T_0(\boldsymbol{\alpha}), \ldots, T_m(\boldsymbol{\alpha}))$. It is defined as $\min_{\boldsymbol{\alpha}} T(\boldsymbol{\alpha})$ subject to the constraints $\alpha_i \geq 0$, $i = 0, \ldots, m$ and $\sum_{i=0}^{m} \alpha_i = 1$. The following theorem proved in [6] characterizes the optimal solution.

Theorem 2.1 (Participation)**.** *In a given linear network, the optimal solution is obtained when all processors participate and they all finish executing their assigned load at the same instant, i.e.,* $T_0(\boldsymbol{\alpha}) = \cdots = T_m(\boldsymbol{\alpha})$.

We introduce the concepts of reduction and equivalent processors used to solve the above problem. *Reduction* is the technique which collapses a set of connected processors and the associated internal links into a single *equivalent processor* that replaces the collapsed processors. Figure 3

Figure 3. The reduction of processors P_i and P_{i+1} to a single equivalent processor.

illustrates a reduction of processors P_i and P_{i+1}. We compute the processing time for the equivalent processor that replaces consecutive processors $(P_i, \ldots, P_{i+s})$ by logically disconnecting the segment from the network and computing the load allocation vector $\boldsymbol{\alpha}$ for it. The *equivalent processing time* $\bar{w}_i$ (*i.e.*, the time to process a unit load by the equivalent processor) is given by

$$\bar{w}_i = \max(T_i(\boldsymbol{\alpha}), \ldots, T_{i+s}(\boldsymbol{\alpha})). \tag{2.3}$$

If $\boldsymbol{\alpha}$ is optimal (*i.e.*, $\boldsymbol{\alpha}$ minimizes $T(\boldsymbol{\alpha})$), $\bar{w}_i$ reduces to

$$\bar{w}_i = T_j(\boldsymbol{\alpha}), \qquad j = i, \ldots, i+s \tag{2.4}$$

Before continuing, we must introduce additional notation.

Notation. Let D_i be the fraction of the original load received by P_i $(i = 0, \ldots, m)$ and let $\hat{\alpha}_i D_i$ $(0 \leq \hat{\alpha}_i \leq 1)$ be the load retained for computing by P_i and $(1 - \hat{\alpha}_i)D_i$ be the load transmitted to its successor. We denote by $\hat{\boldsymbol{\alpha}} = (\hat{\alpha}_0, \ldots, \hat{\alpha}_m)$ the vector of local load allocations as fractions of the received workload. Processor P_m must compute all the received load and thus, $\hat{\alpha}_m = 1$. The relationship between α_i and $\hat{\alpha}_i$ is

$$\alpha_0 = \hat{\alpha}_0 \tag{2.5}$$

$$\alpha_j = \left(\prod_{k=0}^{j-1} (1 - \hat{\alpha}_k)\right) \hat{\alpha}_j, \qquad j = 1, \ldots, m. \tag{2.6}$$

For linear networks with a boundary root comprising more than two processors, we recursively reduce the network, collapsing the two farthest processors from the root at a time until the entire network is represented by a single processor. We derive the following equation for optimal local load allocation for processors P_i and P_{i+1}:

$$\hat{\alpha}_i w_i = (1 - \hat{\alpha}_i)(z_{i+1} + w_{i+1}). \tag{2.7}$$

The following algorithm solves the LINEAR BOUNDARY-LINEAR scheduling problem.

Algorithm 1 (*LINEAR BOUNDARY-LINEAR*)

Input: Processing capacities $w_0, \ldots, w_m$;
Link capacities $z_1, \ldots, z_m$;
Output: Load allocations $\alpha_0, \ldots, \alpha_m$;
1. $\hat{\alpha}_m \leftarrow 1$
2. $\bar{w}_m \leftarrow w_m$
3. **for** $i = m-1, \ldots, 0$; **do**
4. $\hat{\alpha}_i \leftarrow \frac{\bar{w}_{i+1}+z_{i+1}}{w_i+\bar{w}_{i+1}+z_{i+1}}$ by (2.7)
5. $\bar{w}_i \leftarrow \hat{\alpha}_i w_i$ by (2.4)
6. Replace processors P_i and P_{i+1} with a single equivalent processor with processing time $\bar{w}_i$
7. $D \leftarrow 1$
8. **for** $i = 0, \ldots, m$; **do**
9. $\alpha_i \leftarrow D\hat{\alpha}_i$
10. $D \leftarrow D(1-\hat{\alpha}_i)$

In the above algorithm it is assumed that P_i reports its true rate to the mechanism. When the processors are owned and operated by disparate, autonomous organizations that are self-interested and welfare-maximizing, they will misreport their processing capacity or deviate from the algorithm in hope of generating increased profits. In the subsequent sections, we present a mechanism that provides incentives to the agents to report truthfully and that fine agents that deviate from the algorithm.

3. Mechanism Design Framework

In this section, we introduce the main concepts of mechanism design theory. We limit our discussion to mechanisms for one parameter agents. Each agent in this mechanism design problem is characterized by private data represented by a single real value [18]. A *mechanism design problem for one parameter agents* is characterized by

(i) A finite set A of allowed outputs. The output is a vector $\alpha(\mathbf{w}) = (\alpha_1(\mathbf{w}), \ldots, \alpha_m(\mathbf{w})) \in A$, computed according to the agents' bids, $\mathbf{w} = (w_1, \ldots, w_m)$. Here, w_i is the bid of agent i.

(ii) Each agent i ($i = 1, \ldots, m$) has a privately known value t_i called the *true value* and a publicly known parameter $\tilde{w}_i$ called the *actual value*, where $\tilde{w}_i \geq t_i$. The preferences of agent i are given by a function called *valuation* $V_i(\alpha, \tilde{\mathbf{w}})$.

(iii) Each agent goal is to maximize its *utility*. The utility of agent i is $U_i(\mathbf{w}, \tilde{\mathbf{w}}) = Q_i(\mathbf{w}, \tilde{\mathbf{w}}) + V_i(\alpha(\mathbf{w}), \tilde{\mathbf{w}})$, where Q_i is the payment handed by the mechanism to agent i and $\tilde{\mathbf{w}}$ is the vector of actual values. The payments are handed to the agents after the mechanism learns $\tilde{\mathbf{w}}$.

(iv) The goal of the mechanism is to select an output α that optimizes a given cost function $g(\mathbf{w}, \alpha)$.

Definition 3.1 (Mechanism with Verification)**.** *A mechanism with verification is characterized by two functions.*

(i) The output function $\alpha(\mathbf{w}) = (\alpha_1(\mathbf{w}), \ldots, \alpha_m(\mathbf{w}))$. *The input to this function is the vector of agents' bids* $\mathbf{w} = (w_1, \ldots, w_m)$ *and* $\alpha \in A$.

(ii) The payment function $Q(\mathbf{w}, \tilde{\mathbf{w}}) = (Q_1(\mathbf{w}, \tilde{\mathbf{w}}), \ldots, Q_m(\mathbf{w}, \tilde{\mathbf{w}}))$*, where* $Q_i(\mathbf{w}, \tilde{\mathbf{w}})$ *is the payment handed by the mechanism to agent* i.

Notation. In the rest of the paper, we denote by $\mathbf{w}_{-i}$ the vector of bids excluding the bid of agent i. The vector $\mathbf{w}$ is represented by $(\mathbf{w}_{-i}, w_i)$.

The following defines an important property in that an agent will maximize its utility when $\tilde{w}_i = w_i = t_i$ independent of the actions of the other agents.

Definition 3.2 (Strategyproof Mechanism)**.** *A mechanism is called* strategyproof *if for every agent* i *of type* t_i *and for every bids* $\mathbf{w}_{-i}$ *of the other agents, the agent's utility is maximized when it declares its real type* t_i *(i.e., truth-telling is a dominant strategy).*

The next property guarantees non-negative utility for truthful agents. This is important as agents willfully participate in hope of profits.

Definition 3.3 (Voluntary Participation Mechanism)**.** *We say that a mechanism satisfies the* voluntary participation condition *if* $U_i((\mathbf{w}_{-i}, \tilde{\mathbf{w}}_\mathbf{i}) \geq 0$ *for every agent* i*, true value* t_i*, and other agents' bids* $\mathbf{w}_{-i}$ *(i.e., truthful agents never incur a loss).*

There are two models for characterizing distributed mechanisms. They differ in the degree of control that the agents have. A mechanism is a *tamper-proof* mechanism if the agents control the inputs. In these types of mechanism, an agent can only specify its inputs and thus, the only method of cheating is altering its inputs. A more general model is the autonomous node model. A mechanism is an *autonomous node* mechanism if the agents control both the inputs and the algorithm. An agent will implement an algorithm different from what is specified if it is beneficial to do so. In this paper we consider the autonomous node model.

We assume that each processor is characterized by a valuation function which in this case is equal to the cost of processing a given load. A processor wants to maximize its utility which is the sum of its valuation and the payment given to it. A processor P_i is parametrized by its true processing time t_i. It bids its processing time w_i to the mechanism, where w_i may be different than the true processing time t_i. P_i may choose to process its assignment at a different speed than either its true time t_i or bid time w_i. This is its actual processing time $\bar{w}_i$, where $\bar{w}_i \geq t_i$.

4. The Proposed Mechanism

We propose the Divisible Load Scheduling-Linear Bus Linear (DLS-LBL) mechanism for scheduling divisible loads in boundary origination linear networks. The system model comprises $m+1$ processors, where P_0 is the root. The root processor is obedient as it performs tasks on behalf of the mechanism. We model the remaining m processors as strategic nodes. We assume that the communication links are obedient and that the communication protocols are tamper-proof.

Notation. Let SK_i be the private key of a public key set possessed by processor P_i. The secure digital signature of message m under SK_i is $\mathrm{sig}_i(m)$. The message $\mathrm{dsm}_i(m) = (m, \mathrm{sig}_i(m))$ is the digitally signed message m under private key SK_i.

The description of the DLS-LBL mechanism follow. Informally, we assume the existence of a *payment infrastructure* and a *public key infrastructure* (PKI). We assume that all processors have a public cryptographic key set and that the public key from the set is registered with the PKI. Furthermore, we assume a processor P_i knows its predecessor P_{i-1}; the predecessor of P_{i-1}, P_{i-2}; and successor P_{i+1} and it is capable of verifying their signatures.

A processor P_i computes its assigned load in *actual processing time* $\tilde{w}_i$, where $\tilde{w}_i \geq t_i$. We cope with this situation by employing a strategyproof mechanism with verification. The goal of a *strategyproof mechanism with verification* is to give incentives to agents such that it is beneficial for them to report their values and process the assignment using their full capacity. In order to achieve this goal, we augment each processor P_i with a tamper-proof meter that records $\tilde{w}_i$. The meter reports the value as $\mathrm{dsm}_0(\tilde{w}_i)$.

DLS-LBL Mechanism.

Phase I (*Computing local load allocation vector* $\hat{\boldsymbol{\alpha}}$) This phase corresponds to the computation of the vector $\hat{\boldsymbol{\alpha}}$ (steps 1. – 5.) in Algorithm 2 (LINEAR BOUNDARY-LINEAR). Processor P_i $(i = 0, \ldots, m)$ computes its bid $\bar{w}_i$, where $\bar{w}_i$ is the equivalent processing time of processor P_i and its successors. The equivalent processing time $\bar{w}_i$ is given by $\bar{w}_i = \hat{\alpha}_i w_i$ (2.4), where $\hat{\alpha}_m = 1$ and $\hat{\alpha}_j = \frac{\bar{w}_{j+1}+z_{j+1}}{w_j+\bar{w}_{j+1}+z_{j+1}}$ for $j = 0, \ldots, m-1$ (given by (2.7)). Processor P_i $(i = 1, \ldots, m)$ transmits its bid $\mathrm{dsm}_i(\bar{w}_i)$ to its predecessor P_{i-1}. We denote by $\bar{\mathbf{w}} = (\bar{w}_0, \ldots, \bar{w}_m)$ the vector of bids. Processor P_{i-1} terminates the protocol if it does not receive a message, receives malformed or inauthentic messages, or receives contradictory messages. Messages are contradictory when two or more authentic messages having different contents are received from a sender. In the event that processor P_{i-1} receives contradictory messages, the evidence is submitted to P_0. Processor P_0 penalizes P_i with a fine of F and rewards it to P_{i-1} if the claim is substantiated. The quantity F must be larger than any potential profits attainable by cheating. If P_0 exculpates P_i, P_{i-1} is fined F and P_i is rewarded F.

Phase II (*Computing load allocation vector* $\boldsymbol{\alpha}$) We compute the load allocation vector $\boldsymbol{\alpha}$ from the local load allocation vector $\hat{\boldsymbol{\alpha}}$ computed in the previous phase. This phase corresponds to steps 7. – 10. of Algorithm 1. Processor P_0 sends the message

$$G_1 = (\,\mathrm{dsm}_0(D_0), \mathrm{dsm}_0(D_1), \mathrm{dsm}_0(\bar{w}_0), \mathrm{dsm}_0(w_0), \mathrm{dsm}_0(\bar{w}_1)) \tag{4.1}$$

to P_1 and processor P_{i-1} $(i = 2, \ldots, m)$ transmits the message

$$G_i = (\,\mathrm{dsm}_{i-2}(D_{i-1}), \mathrm{dsm}_{i-1}(D_i), \mathrm{dsm}_{i-2}(\bar{w}_{i-1}), \mathrm{dsm}_{i-1}(w_{i-1}), \mathrm{dsm}_{i-1}(\bar{w}_i)) \tag{4.2}$$

to successor P_i, where D_j $(j = 0, \ldots, m)$ is the quantity of load received by processor P_j defined as $D_0 = 1$ (root must handle the entire initial load) and $D_j = \prod_{h=0}^{j-1}(1-\hat{\alpha}_h)$ for $j = 1, \ldots, m$. Processor P_i verifies the message authenticity and integrity and it terminates the protocol if either check fails. It verifies that its bid $\mathrm{dsm}_{i-1}(\bar{w}_i)$ is contained within the message and that $\bar{w}_{i-1} = \hat{\alpha}_{i-1} w_{i-1}$ and $\hat{\alpha}_{i-1} w_{i-1} = (1-\hat{\alpha}_{i-1})(w_i + z_i)$, where $\hat{\alpha}_{i-1} = \frac{D_{i-1}-D_i}{D_{i-1}}$. Again, it terminates the protocol if the checks fail. If the termination is due to the reception of contradictory messages or incorrect computations, P_i sends the evidence to P_0. The root fines P_{i-1} a sum of F and rewards it to P_i if the root can substantiate the claim. Otherwise, processor P_i is penalized F which is rewarded to P_{i-1}. In either case, processors not partaking in complaints receive zero utility. Processor P_i computes its load allocation $\alpha_i = D_i \hat{\alpha}_i$.

Phase III (*Load distribution and computation*) The load is distributed from processor to processor until all processors receive their assignment. Beginning with the root, processor P_i $(i = 0, \ldots, m-1)$ distributes $1-\hat{\alpha}_i$ work units to its successor P_{i+1}; processor P_i retains $\hat{\alpha}_i$ work units for itself to compute. In order to increase its utility (we disclose the reasons shortly), a processor P_i may deviate from α_i by retaining $\tilde{\alpha}_i$ work units, where $0 \leq \tilde{\alpha}_i < \alpha_i$. Let $\hat{\tilde{\alpha}}_i$ be the *actual local load allocation* which corresponds to $\tilde{\alpha}_i$. P_i will distribute $1-\hat{\tilde{\alpha}}_i$ fractions of work to it successor P_{i+1} and thus, increasing the successors' work load. To combat this scenario, we assume that the data is embedded with a

device Λ_i that permits processor P_i to prove it received no more than Λ_i work units[1]. When a processor P_{i+1} receives too much work (*i.e.*, $D_i\hat{\alpha}_i > \alpha_i$), it itself computes the additional $\tilde{\alpha}_i - \alpha_i$ units. When processing is completed, processor P_{i+1} notifies the root of receiving additional load. It supports its claim by submitting $Grievance_{i+1} = (G_{i+1}, \Lambda_{i+1}, \mathrm{dsm}_0(\tilde{w}_i))$. If the claim is valid, offender P_i is penalized the sum $F + (\tilde{\alpha}_{i+1} - \alpha_{i-1})\tilde{w}_{i+1}$ and the victim P_{i+1} is rewarded F. In the next phase, the mechanism compensates P_{i+1} the amount $(\tilde{\alpha}_{i+1} - \alpha_{i-1})\tilde{w}_{i+1}$ for the additional work it performed.

Phase IV (*Payment computation*) Processor P_i ($i = 0, \ldots, m$) computes its own payment. Processor P_0 behaves obediently and thus does not require a bonus to obey the mechanism. The mechanism reimburses processor P_0 for the work it performed. The *utility* U_0 of P_0 is

$$U_0(\alpha_0, \tilde{w}_0) = V_0(\alpha_0, \tilde{w}_0) + C_0(\alpha_0, \tilde{w}_0), \qquad (4.3)$$

where $V_0(\alpha_0, \tilde{w}_0) = -\alpha_0\tilde{w}_0$ and $C_0(\alpha_0, \tilde{w}_0) = \alpha_0\tilde{w}_0$. Therefore, $U_0 = 0$. The goal of processor P_j ($j = 1, \ldots, m$) is to maximize its utility. The utility U_j of processor P_j is

$$U_j = V_j(\tilde{\alpha}_j, \tilde{w}_j) + Q_j(\alpha_j, \tilde{\alpha}_j, w_{j-1}, \bar{w}_j, w_j, \tilde{w}_j) \qquad (4.4)$$

where

$$V_j(\tilde{\alpha}_j, \tilde{w}_j) = -\tilde{\alpha}_j\tilde{w}_j \qquad (4.5)$$

is the *valuation function*. The *payment function* Q_j is

$$Q_j(\alpha_j, \tilde{\alpha}_j, w_{j-1}, \bar{w}_j, w_j, \tilde{w}_j) = \begin{cases} 0 & \text{if } \tilde{\alpha}_j = 0, \\ C_j(\alpha_j, \tilde{\alpha}_j, \tilde{w}_j) + B_j(\alpha_j, w_{j-1}, \bar{w}_j, w_j, \tilde{w}_j) & \text{if } \tilde{\alpha}_j > 0, \end{cases} \qquad (4.6)$$

where

$$C_j(\alpha_j, \tilde{\alpha}_j, \tilde{w}_j) = \alpha_j\tilde{w}_j + E_j(\alpha_j, \tilde{\alpha}_j\tilde{w}_j) \qquad (4.7)$$

is the *compensation function* and

$$E_j(\alpha_j, \tilde{\alpha}_j, \tilde{w}_j) = \begin{cases} 0 & \text{if } \tilde{\alpha}_j < \alpha_j, \\ (\tilde{\alpha}_j - \alpha_j)\tilde{w}_j & \text{if } \tilde{\alpha}_j \geq \alpha_j, \end{cases} \qquad (4.8)$$

is the *recompense function* that reimburses overloaded processors for performing additional work. The *bonus function* is

$$B_j(\alpha_j, w_{j-1}, \bar{w}_j, w_j, \tilde{w}_j) = w_{j-1} - \bar{w}_{j-1}(\boldsymbol{\alpha}((w_{j-1}, \bar{w}_j)), (w_{j-1}, \hat{w}_j)). \qquad (4.9)$$

The function $\bar{w}_{j-1}(\boldsymbol{\alpha}((w_{j-1}, \bar{w}_j)), (w_{j-1}, \hat{w}_j))$ is the processing time of the equivalent processor comprising P_{j-1} and its successors adjusted for the actual processing time of P_j, where

$$\hat{w}_m = \tilde{w}_m \qquad (4.10)$$

and

$$\hat{w}_k = \begin{cases} \hat{\alpha}_k\tilde{w}_k & \text{if } \tilde{w}_k \geq w_k, \\ \bar{w}_k & \text{if } \tilde{w}_k < w_k, \end{cases} \qquad (4.11)$$

for $k = 1, \ldots, m-1$. The processing time $\hat{w}_j$ is the bid time of the equivalent processor comprising P_j and its successors adjusted for the actual performance of P_j. The time $\hat{w}_j$ is dominated by the performance of processor P_j when it runs slower than bid ($\tilde{w}_j > w_j$); if P_j runs faster ($\tilde{w}_i < w_i$), the equivalent processing time remains unchanged. Processor P_i saves

$$Proof_j = (G_j, \mathrm{dsm}_{j+1}(\bar{w}_{j-1}), \mathrm{dsm}_j(w_j), \mathrm{dsm}_0(\tilde{w}_j), \Lambda_j) \qquad (4.12)$$

as evidence of correct payment computation. Processor P_j submits bill Q_j to the payment infrastructure. With probability q, where $0 < q \leq 1$, the root requests $Proof_j$ from P_j. If P_j fails to provide a valid proof, it is penalized F/q.

This concludes the descriptions of the DLS-LBL mechanism. The mechanism as described is valid for selfish-but-agreeable agents but not for selfish-and-annoying agents. A *selfish-but-agreeable* agent will deviate from the algorithm only if it strictly improves its welfare, while a *selfish-and-annoying* agent will only follow the prescribed algorithm if it is the only action that maximizes its welfare. The selfish-and-annoying processors will subvert the mechanism by performing undesirable actions (*e.g.*, corrupting data, sending the same data set to multiple children, etc.) where their behavior is not constrained by incentives or penalties. If the load is associated with a problem where the solution can be verified (*e.g.*, searches, factorizations), we can easily amend the mechanism to tolerate selfish-and-annoying processors. We begin by altering (4.6) to

$$Q_j(\alpha_j, \tilde{\alpha}_j, w_{j-1}, \bar{w}_j, w_j, \tilde{w}_j) = \begin{cases} 0 & \text{if } \tilde{\alpha}_j = 0, \\ C_j(\alpha_j, \tilde{\alpha}_j, \tilde{w}_j) + \\ B_j(\alpha_j, w_{j-1}, \bar{w}_j, w_j, \tilde{w}_j) + S & \text{if } \tilde{\alpha}_j > 0, \end{cases} \qquad (4.13)$$

[1] Data preparation is an example of a simple Λ_i. We divide the data into equal-sized blocks and then append to each a unique, random identifier. The identifier space must be large enough so that the probability of an agent successfully guessing a valid identifier is small. Submitting the identifiers allows P_i to show the amount of data it received.

where S is the *solution bonus*. $S = 0$ if a solution is not found and $S = s$ if a solution is found. The bonus s is a small, positive quantity that rewards agents for following the given algorithm. Selfish-and-annoying agents will not risk the loss of s; hence, they will not deviate from the prescribed algorithm.

5. DLS-LBL Properties

In this section we study the properties of DLS-LBL. We first prove the strategyproofness of the mechanism.

Lemma 5.1. A selfish-but-agreeable processor will be fined for deviating from the DLS-LBL mechanism.

Proof. Let P_i be a selfish-but-agreeable agent. A selfish-but-agreeable agent will deviate from the algorithm if the action is beneficial, *i.e.*, $U_i' > U_i$, where U_i' is the utility of a deviating P_i. Processor P_i may deviate from the algorithm by: (i) sending contradictory messages in Phase I or II, (ii) incorrectly computing $\bar{w}_i$ in Phase I or D_{i+1} in Phase II, (iii) decreasing its work load ($\tilde{\alpha}_i < \alpha_i$) and thus increasing its successors' work loads ($D_i(1 - \tilde{\alpha}_i) > D_i(1 - \alpha_i)$) in Phase III, (iv) overcharging in Phase IV, or (v) falsely accusing another of cheating in Phase I, II, and III. Processor P_i will not deviate in other fashions (*e.g.*, corrupting data) because there is no benefits to do so. We combat these situations by rewarding processors who report deviants. In any instance that a deviant is caught, it is penalized a sum greater than any profits attainable by cheating. We now show that for each case, the mechanism detects cheating processors. In case (i), the recipient will report P_i. In case (ii), the successor P_{i+1} validates the values in message G_{i+1}. If inconsistencies are discovered, P_{i+1} reports P_i. In case (iii), successor P_{i+1} reports P_i for receiving the additional load. In case (iv), the fine F/q, where $0 < q \leq 1$ is the probability of challenge, is the deterrent for overcharging. The complete proof for case (iv) can be found in [17]. In case (v), processor P_i does not have the evidence to substantiate its claim and thus it is fined. □

Lemma 5.2. A processor receives a fine only if it has deviated from DLS-LBL.

Proof. Processor P_i is fined for either deviating from the protocol or another processor P_j ($i \neq j$) produces contradictory messages signed by P_i. In the first case, P_i clearly deviates from the algorithm. In the second case, P_j signs the messages either by successfully forging P_i's signature or by possessing private key SK_i. We assume that the forging of signatures is impossible. Processor P_j obtains SK_i either by P_i sharing it or by stealing it from P_i. It is a violation of the mechanism for a second party to possess SK_i. Thus, P_i is fined for protocol deviation. □

Theorem 5.1. *(Selfish-but-Agreeable Agent Compliance)* A selfish-but-agreeable processor does not have incentives to deviate from DLS-LBL.

Proof. Following from Lemma 5.1 and 5.2, a selfish-but-agreeable processor P_i will be fined for and only for deviating. The fine is larger than any profits attainable by cheating and thus will abate any willingness to cheat. Therefore, the processor P_i will not deviate. □

Theorem 5.2. *(Selfish-and-Annoying Agent Compliance)* A selfish-and-annoying agent does not have incentives to deviate from DLS-LBL if the solution bonus function is employed.

Proof. Let processor P_i be a selfish-and-annoying agent. Theorem 5.2 handles the cases in which deviation is beneficial, *i.e.*, $U_i' > U_i$, where U_i' is the utility of the deviating P_i. Processor P_i will deviate as long as there is no reduction in utility, *i.e.*, $U_i' = U_i$. Examples include P_i corrupting data or sending the same data to different children. These actions reduce the probability of obtaining a solution and thus, reduce the probability of receiving the solution bonus. Processor P_i is welfare maximizing; hence, it will not choose to perform such actions. Therefore, processor P_i does not have incentives to deviate from the mechanism. □

Lemma 5.3. The mechanism is strategyproof if the processors do not deviate from the algorithm.

Proof. The utility U_i of processor P_i is

$$\begin{aligned} U_i = V_i + Q_i = -\tilde{\alpha}_i \tilde{w}_i + \alpha_i \tilde{w}_i + (\tilde{\alpha}_i - \alpha_i)\tilde{w}_i + \\ w_{i-1} - \bar{w}_{i-1}(\boldsymbol{\alpha}((w_{i-1}, \bar{w}_i)), (w_{i-1}, \hat{w}_i)). \end{aligned} \tag{5.1}$$

We assume that the processors do not deviate from the algorithm and thus, abide by the computed load allocation, *i.e.*, $\tilde{\alpha}_i = \alpha_i$. The utility U_i is

$$U_i = w_{i-1} - \bar{w}_{i-1}(\boldsymbol{\alpha}((w_{i-1}, \bar{w}_i)), (w_{i-1}, \hat{w}_i)). \tag{5.2}$$

We consider two cases:

(i) $\tilde{w}_i = t_i$, *i.e.*, processor P_i computes the load at full capacity. Assume P_i is a terminal processor. If P_i bids its true value $w_i^e = t_i$, then its utility U_i^e is

$$\begin{aligned} U_i^e &= w_{i-1} - \bar{w}_{i-1}(\boldsymbol{\alpha}((w_{i-1}, t_i)), (w_{i-1}, t_i)) \\ &= w_{i-1} - \bar{w}_{i-1}^e. \end{aligned} \tag{5.3}$$

If P_i bids lower ($w_i^l < t_i$), then its utility U_i^l is

$$\begin{aligned} U_i^l &= w_{i-1} - \bar{w}_{i-1}(\boldsymbol{\alpha}((w_{i-1}, w_i^l)), (w_{i-1}, t_i)) \\ &= w_{i-1} - \bar{w}_{i-1}^l. \end{aligned} \tag{5.4}$$

We want to show $U_i^e \geq U_i^l$, which reduces to showing $\bar{w}_{i-1}^e \leq \bar{w}_{i-1}^l$. By the LINEAR BOUNDARY-LINEAR algorithm, we know that $\boldsymbol{\alpha}((w_{i-1}, t_i))$ is optimal. By bidding lower than the true vale, P_i is assigned more load and the other processors are assigned less load. The greater load will increase the execution time of P_i and increase the equivalent processing rate such that $\bar{w}_{i-1}^e \leq \bar{w}_{i-1}^l$. Therefore, $U_i^e \geq U_i^l$. The other possibility is that P_i bids higher ($w_i^h > t_i$). Its utility U_i^h is

$$\begin{aligned} U_i^h &= w_{i-1} - \bar{w}_{i-1}(\boldsymbol{\alpha}((w_{i-1}, w_i^h)), (w_{i-1}, t_i)) \\ &= w_{i-1} - \bar{w}_{i-1}^h. \end{aligned} \quad (5.5)$$

Similar to above, we want to show $U_i^e \geq U_i^h$. Bidding higher than the true value results in reduced load to P_i and increased load to the other processors. Since $\boldsymbol{\alpha}((w_{i-1}, t_i))$ is optimal, $\bar{w}_{i-1}^e \leq \bar{w}_{i-1}^h$ and thus, $U_i^e \geq U_i^h$.

We now assume P_i to be an interior processor. If P_i bids its true value ($w_i^e = t_i$), then its utility U_i^e is

$$\begin{aligned} U_i^e &= w_{i-1} - \bar{w}_{i-1}(\boldsymbol{\alpha}((w_{i-1}, \bar{w}_i^e)), (w_{i-1}, \bar{w}_i^e)) \\ &= w_{i-1} - \bar{w}_{i-1}(\boldsymbol{\alpha}((w_{i-1}, \bar{w}_i^e)), (w_{i-1}, \hat{\alpha}_i^e t_i)) \\ &= w_{i-1} - \bar{w}_{i-1}^e \end{aligned} \quad (5.6)$$

where $\bar{w}_i^e$ is the processing rate of equivalent processor P_i. If P_i bids lower ($w_i^l \leq t_i$), then its utility U_i^l is

$$\begin{aligned} U_i^l &= w_{i-1} - \bar{w}_{i-1}(\boldsymbol{\alpha}((w_{i-1}, \bar{w}_i^l)), (w_{i-1}, \hat{\alpha}_i^l w_i^l)) \\ &= w_{i-1} - \bar{w}_{i-1}^l \end{aligned} \quad (5.7)$$

where $\bar{w}_i^l$ is the equivalent processing rate of P_i. We know that $\boldsymbol{\alpha}((w_{i-1}, \bar{w}_i^e))$ is the optimal allocation by the LINEAR BOUNDARY-LINEAR algorithm. By bidding lower, P_i is assigned more load, *i.e.*, $\alpha_i^l \geq \alpha_i^e$. The performance of the network is constrained by P_i. Thus, $\bar{w}_{i-1}^e \leq \bar{w}_{i-1}^l$ which proves $U_i^e \geq U_i^l$. Finally, if P_i bids higher ($w_i^h \geq t_i$), then its utility U_i^h is

$$\begin{aligned} U_i^h &= w_{i-1} - \bar{w}_{i-1}(\boldsymbol{\alpha}((w_{i-1}, \bar{w}_i^h)), (w_{i-1}, \bar{w}_i)) \\ &= w_{i-1} - \bar{w}_i^h. \end{aligned} \quad (5.8)$$

where $\bar{w}_i^h = \alpha_i^h w_i^h$. We know that $\boldsymbol{\alpha}((w_{i-1}, \bar{w}_i^e))$ is the optimal allocation. By bidding higher, less load is assigned to P_i and more load is assigned to the other processors thus reducing the performance. This results in $\bar{w}_i^e \leq \bar{w}_i^h$; hence, $U_i^e \geq U_i^h$.

(ii) $\tilde{w}_i > t_i$, *i.e.*, processor P_i computes the load slower than its full processing capacity. A similar argument as in case (i) applies. □

Theorem 5.3. *(Strategyproofness)* The DLS-LBL mechanism is strategyproof.

Proof. Lemma 5.3 states that the mechanism is strategyproof as long as the processors do not deviate. The processors, by Theorem 5.1 and 5.2, do not have incentives to deviate. Therefore, the mechanism is strategyproof. □

We now show that the mechanism satisfies the voluntary participation condition.

Lemma 5.4. If the processors do not deviate from the protocol, the DLS-LBL mechanism satisfies the voluntary participation condition.

Proof. The utility U_i ($i = 1, \ldots, m-1$) of an interior processor P_i when it bids its true value is

$$U_i = w_{i-1} - \bar{w}_{i-1}(\boldsymbol{\alpha}((w_{i-1}, \bar{w}_i)), (w_{i-1}, \hat{\alpha}_i t_i)). \quad (5.9)$$

The utility U_m of the terminal processor P_m when it bids its true value is

$$\begin{aligned} U_m = w_{m-1} - \\ \bar{w}_{m-1}(\boldsymbol{\alpha}((w_{m-1}, \bar{w}_m)), (w_{m-1}, t_m)). \end{aligned} \quad (5.10)$$

The load allocation $\boldsymbol{\alpha}((w_{j-1}, \bar{w}_j))$, for $j = 1, \ldots, m$, is optimal. We know that $\bar{w}_{j-1} = \hat{\alpha}_{j-1} w_{j-1}$, where $0 < \hat{\alpha}_{j-1} \leq 1$. Therefore, $\bar{w}_{j-1} \leq w_{j-1}$ and $U_j \geq 0$. □

Theorem 5.4. (*Voluntary Participation*) The DLS-LBL mechanism satisfies the voluntary participation condition.

Proof. Lemma 5.4 states that the mechanism satisfy the voluntary participation condition as long as no deviation occurs. We know by Theorem 5.1 and 5.2 that processors are unwilling to deviate. Therefore, the mechanism satisfies the voluntary participation condition. □

6. Conclusion

In this paper we proposed a strategyproof mechanism, DLS-LBL, for scheduling divisible loads in linear networks. Load origination in a linear network occurs at the root processor. It is either a terminal processor or an interior processor. The DLS-LBL mechanism schedules loads when the root is a terminal processor. Through the use of incentives, processors report their true parameters and process their assignments at full capacity. Additional incentives are provided for reporting processors that deviate from the algorithm. A processor will readily report a deviant in order to receive a reward. The deviants are penalized a sum greater than any profits attainable by cheating, which dissuades them from attempting it. Besides being strategyproof, the mechanism also satisfies the voluntary participation condition. All truthful processors will obtain non-negative utility and thus will participate in hope of profits.

Our plan for future work is to propose and study mechanisms for different network architectures under various assumptions. The goal is to achieve a cohesive theory combining DLT with incentives.

References

[1] A. Archer and E. Tardos. Truthful mechanism for one-parameter agents. In *Proc. of the 42nd IEEE Symp. on Foundations of Computer Science*, pages 482–491, Oct. 2001.

[2] O. Beamount, L. Marchal, V. Rehn, and Y. Robert. FIFO scheduling of divisible loads with return messages under the one-port model. In *Proc. of the 20th IEEE International Parallel and Distributed Processing Symp.*, Apr. 2006.

[3] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang. Scheduling divisible loads on star and tree networks: Results and open problems. *IEEE Trans. Parallel and Distributed Syst.*, 16(3):207–218, Mar. 2005.

[4] W. Bethel, B. Tierney, J. Lee, D. Gunter, and S. Lau. Using high-speed WANs and network data caches to enable remote and distributed visualization. In *SC2000: High Performance Networking and Computing*, Nov. 2000.

[5] V. Bharadwaj and G. Barlas. Access time minimization for distributed multimedia applications. *Multimedia Tools and Applications*, 12(2-3):235–256, Nov. 2000.

[6] V. Bharadwaj, D. Ghose, V. Mani, and T. G. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[7] V. Bharadwaj, D. Ghose, and T. G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–17, Jan. 2003.

[8] J. Blazewicz, M. Drozdowski, and M. Markiewicz. Divisible task scheduling - concept and verification. *Parallel Computing*, 25(1):87–98, Jan. 1999.

[9] T. E. Carroll and D. Grosu. A strategyproof mechanism for scheduling divisible loads in tree networks. In *Proc. of the 20th IEEE International Parallel and Distributed Processing Symp.*, Apr. 2006.

[10] S. Chan, V. Bharadwaj, and D. Ghose. Large matrix-vector products on distributed bus networks with communication delays using the divisible load paradigm: Performance and simulation. *Mathematics and Computers in Simulation*, 58:71–92, 2001.

[11] J. Feigenbaum, C. Papadimitriou, R. Sami, and S. Shenker. A BGP-based mechanism for lowest-cost routing. In *Proc. of the 21st ACM Symp. on Principles of Distributed Computing*, pages 173–182, July 2002.

[12] J. Feigenbaum, C. H. Papadimitriou, and S. Shenker. Sharing the cost of multicast transmissions. *Journal of Computer and System Sciences*, 63(1):21–41, Aug. 2001.

[13] J. Feigenbaum and S. Shenker. Distributed algorithmic mechanism design: Recent results and future directions. In *Proc. of the 6th ACM Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 1–13, Sept. 2002.

[14] D. Grosu and T. E. Carroll. A strategyproof mechanism for scheduling divisible loads in distributed systems. In *Proc. of the 4th International Symp. on Parallel and Distributed Computing*, pages 83–90. IEEE Computer Society Press, July 2005.

[15] D. Grosu and A. T. Chronopoulos. Algorithmic mechanism design for load balancing in distributed systems. *IEEE Trans. Systems, Man and Cybernetics - Part B: Cybernetics*, 34(1):77–84, Feb. 2004.

[16] X. Li, V. Bharadwaj, and C. Ko. Distributed image processing on a network of workstations. *Intl. Journal of Computers and Their Applications*, 25(2):1–10, 2003.

[17] J. C. Mitchell and V. Teague. Autonomous nodes and distributed mechanisms. In *Proc. of the Mext-NSF-JSPS International Symp. on Software Security - Theories and Systems*, pages 58–83, Nov. 2003.

[18] N. Nisan and A. Ronen. Algorithmic mechanism design. *Games and Economic Behaviour*, 35(1/2):166–196, Apr. 2001.

[19] T. G. Robertazzi. Ten reasons to use divisble load theory. *IEEE Computer*, 36(5):63–68, May 2003.

[20] W. E. Walsh, M. P. Wellman, P. R. Wurman, and J. K. MacKie-Mason. Some economics of market-based distributed scheduling. In *Proc. of the 18th IEEE International Conference on Distributed Computing Systems*, pages 612–621, May 1998.

[21] Y. Yang, K. van der Raadt, and H. Casanova. Multiround algorithms for scheduling divisible loads. *IEEE Trans. Parallel and Distributed Syst.*, 16(11):1092–1102, Nov. 2005.