A Strategyproof Mechanism for Scheduling Divisible Loads in Tree Networks

Thomas E. Carroll and Daniel Grosu

Wayne State University
Department of Computer Science
5143 Cass Avenue, Detroit, MI 48202, USA.
{tec, dgrosu}@cs.wayne.edu

Abstract

The underlying assumption of Divisible Load Scheduling is that the processors composing the network are obedient, i.e., they do not "cheat" the algorithm. This assumption is unrealistic if the processors are owned by autonomous, self-interested organizations that have no a priori *motivation for cooperation and they will manipulate the algorithm if it is beneficial to do so. In this paper we propose the strategyproof mechanism DLS-TL for scheduling divisible loads in tree networks. Our proposal augments Divisible Load Theory (DLT) with incentives such that it is beneficial for processors to report their true processing capacity and compute their assignments at full processing capacity. Additionally, incentives are provided for processors to report algorithm deviants. Deviants are penalized which abates the processors' willingness to deviate.*

1. Introduction

One of the most studied topics in distributed systems is scheduling. Poor scheduling decisions lead to inefficiencies, underutilized resources, and suboptimal performance. In this paper we focus on the problem of scheduling divisible loads. Divisible load problems are characterized by data parallelism. These problems have 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. These problems commonly arise in many domains including image processing [15], databases [8], linear algebra [9], visualization [4], and multimedia broadcasting [5].

Scheduling divisible loads is the subject of Divisible Load Theory (DLT) which was extensively studied in [6] where influences such as network architectures (*e.g.*, linear, bus, tree), task arrangements, and optimality conditions are explored. The underlying assumption of DLT is that the processors are *obedient*, *i.e.*, under no circumstances will the processors "cheat". In the real world, the assumption is unrealistic as the nodes may be owned by autonomous, self-interested entities that have no *a priori* motivation for cooperation and they are tempted to manipulate the algorithms in hope of increased benefits. In this type of environment, the processors are properly modeled as *strategic* agents. New protocols for DLT must account for this self-interested behavior. *Mechanism design theory* [21] — a field of economics that has recently garnered interest in computer science — provides the framework for solving such problems involving self-interested parties. The theory addresses *incentive compatibility*: rational agents (self-interested, utility-maximizing) are provided incentives which induce a behavior that maximizes the social welfare. Of interest are the strategyproof mechanisms. Each participant in a mechanism is characterized by private parameters. A *strategyproof mechanism* will result in a participant maximizing its utility if it truthfully reports its private parameters and follows the specified algorithm.

In our previous work [13], we showed how DLT can be augmented with incentives. We designed the strategyproof DLS-BL mechanism for scheduling divisible loads in *bus networks*. DLS-BL provides incentives to the processors to participate and to report their processing capacity to the centralized, trusted scheduler. The agents maximize their welfare by truthful reporting their values to the mechanism and executing their assignments as reported.

In this paper we look to augment DLT with incentives for *tree networks*, where the network comprises strategic processors. In this model, the load is distributed from the root of the tree downward through intermediate processors until all processors are assigned load. We propose the strategyproof mechanism DLS-TL to optimally distribute load among the processors in a tree network. The mechanism provides incentives to the processors to participate and report their full processing capacity. The DLS-TL is an example of *autonomous node mechanism*, 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 a different algorithm if it is beneficial to do such. To combat this scenario, processors are provided incentives to report algorithm deviants. Penalties abate processors' willingness to deviate.

Related work. The divisible load scheduling problem was studied extensively in recent years resulting in a cohesive theory called Divisible Load Theory (DLT). A reference book on DLT is [6]. Two recent surveys on DLT are [7] and [22]. This theory has been used for scheduling loads on heterogeneous distributed systems in the context of different applications such as image processing [15], databases [8], linear algebra [9], and multimedia broadcasting [5]. Scheduling divisible loads in grids has been investigated in [25]. New results and open research problems in DLT are presented in [3]. All these works assumed that the participants in the load scheduling algorithms are obedient and follow the algorithm. Recently, several researchers considered the mechanism design theory to solve several computational problems that involve self-interested participants. These problems include resource allocation and task scheduling [19, 23, 24], routing [10] and multicast transmission [11]. In their seminal paper, Nisan and Ronen [20] 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 task scheduling on unrelated machines problem. 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. In a subsequent paper [2] the same authors investigated the frugality of shortest path mechanisms. Grosu and Chronopoulos [14] derived a strategyproof mechanism that gives the overall optimal solution for the static load balancing problem in distributed systems. A strategyproof mechanism with verification combining incentives and DLT was proposed by Grosu and Carroll [13]. The results and the challenges of designing distributed mechanisms are surveyed in [12]. The *strategyproof computing* paradigm proposed in [17] considers the self-interest and incentives of participants in distributed computing systems. Ng *et al.* [18] proposed a strategyproof system for dynamic resource allocation in data staging. Mitchell and Teague [16] extended the distributed mechanism in [11] devising a new model where the agents themselves implement the mechanism, thus allowing them to deviate from the algorithm.

Our contributions. The main contribution of this paper is the design of a strategyproof mechanism with verification for scheduling divisible loads in tree networks assuming a linear cost model for the processors. We define the mechanism and prove its properties.

Organization. The paper is structured as follows. In Section 2 we present a description of the divisible load scheduling problem in the context of tree networks. In Section 3 we discuss the mechanism design foundations. In Section 4 we present our proposed mechanism. In Section 5 we prove the mechanism's properties. In Section 6 we draw conclusions and present future directions.

2. Divisible Load Scheduling Problem

We first consider a distributed system comprising $m+1$ processors interconnected in a *single-level tree network*. The network is composed of a set of terminal processors, $(P_1,\dots,P_m)$, and their parent, P_0. Processor P_i $(i=0,\dots,m)$ 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 the assignment. This corresponds to a linear cost model. The parent P_0 is the *load-originating processor* that is responsible for distributing the load to the children. We assume that the processors have front-ends that allow simultaneous communication and processing and that the root processor can communicate with only one terminal processor at a given time (*i.e.*, we assume the one-port model). Processor P_0 transmits α_j $(j=1,\dots,m)$ units of load to child P_j in time $\alpha_j z_j$, where z_j is the time it takes to communicate a unit load from P_0 to child P_j. We denote by $\boldsymbol{\alpha}=(\alpha_0,\dots,\alpha_m)$ the vector of load allocations. Processor P_i finishes its assignment in time $T_i(\boldsymbol{\alpha})$, which is the total time to receive (if $i\neq 0$) and process its assignment. The execution on a single-level tree is depicted in Figure 1.

The scheduling problem denoted as SINGLE-LEVEL TREE-LINEAR is to determine the optimal load allocation $\boldsymbol{\alpha}$ which minimizes the total execution time $T(\boldsymbol{\alpha}) = \max(T_0(\boldsymbol{\alpha}),\dots,T_m(\boldsymbol{\alpha}))$. The finish time $T_i(\boldsymbol{\alpha})$ of processor P_i is

$$T_i(\boldsymbol{\alpha}) = \begin{cases} 0 & \text{if } \alpha_i = 0 \\ \alpha_i w_i & \text{if } i=0 \text{ and } \alpha_i > 0 \\ \sum_{j=1}^{i} \alpha_j z_j + \alpha_i w_i & \text{if } i\neq 0 \text{ and } \alpha_i > 0. \end{cases} \quad (1)$$

The SINGLE-LEVEL TREE-LINEAR problem can be formalized as

$$\min_{\boldsymbol{\alpha}} T(\boldsymbol{\alpha}) \quad (2)$$

subject to constraints: (i) $\alpha_i \geq 0, i=0,\dots,m$, and (ii) $\sum_{i=0}^{m} \alpha_i = 1$

The following theorems proved in [6] characterize the optimal solution.

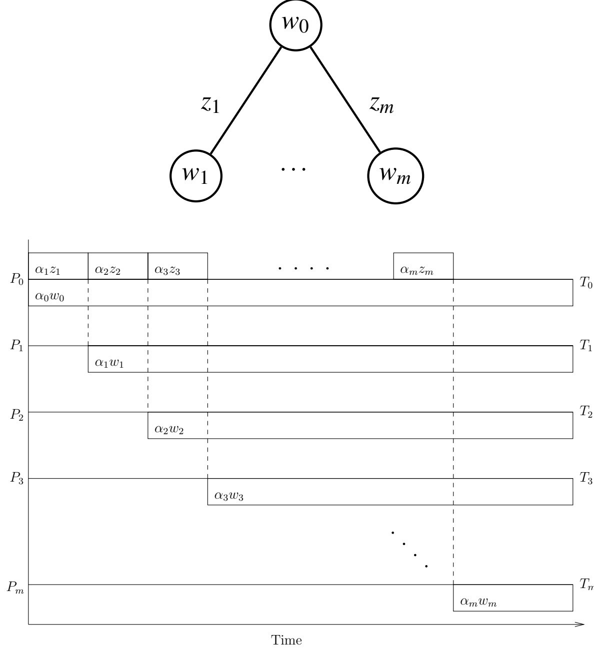

Figure 1. Execution on a $m+1$ single-level tree.

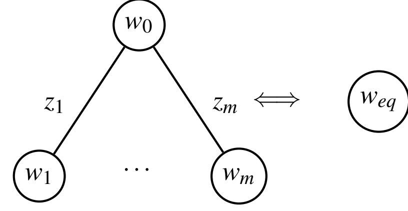

Figure 2. The reduction of a single-level tree to a single equivalent processor

Theorem 2.1. *(Sequencing)* An optimal solution to the SINGLE-LEVEL TREE-LINEAR problem is obtained when $z_1 \leq z_2 \leq \ldots \leq z_m$.

Theorem 2.2. *(Participation)* The optimal solution for the SINGLE-LEVEL TREE-LINEAR problem is obtained when all processors participate and they all finish executing their assigned load at the same time, *i.e.*, $T_0(\boldsymbol{\alpha}) = T_1(\boldsymbol{\alpha}) = \ldots = T_m(\boldsymbol{\alpha})$.

From Figure 1, we derive the following recursive equations for the optimal load allocation

$$\alpha_0 w_0 = \alpha_1 z_1 + \alpha_1 w_1 \tag{3}$$

$$\alpha_j w_j = \alpha_{j+1} z_{j+1} + \alpha_{j+1} w_{j+1}, \qquad j = 1, \ldots, m-1. \tag{4}$$

We can reduce the single-level tree to a single processor with equivalent processing capacity. Figure 2 illustrates the reduction. We compute the *equivalent processing capacity* w_{eq} by

$$w_{eq} = \max(T_0(\boldsymbol{\alpha}), \ldots, T_m(\boldsymbol{\alpha})). \tag{5}$$

Time T_i is the time P_i takes to complete its portion of the unit load. Since $\sum_{j=0}^{m} \alpha_j = 1$, the processing capacity of the tree is equal to the finish time of the slowest processor. If $\boldsymbol{\alpha}$ is optimal (*i.e.*, $\boldsymbol{\alpha}$ resulted from (3) and (4)), then w_{eq} reduces to

$$w_{eq} = T_i(\boldsymbol{\alpha}), \qquad i = 0, \ldots, m. \tag{6}$$

The subsequent algorithm solves the SINGLE-LEVEL TREE-LINEAR problem.

Algorithm 2.1. (*SINGLE-LEVEL TREE-LINEAR*)

Input: Processing capacities $w_0, \ldots, w_m$;
Link capacities $z_1, \ldots, z_m$ such that $z_1 \leq \ldots \leq z_m$;
Output: Load allocations $\alpha_0, \ldots, \alpha_m$;
Equivalent processing capacity w_{eq};
1. Compute $\alpha_0, \ldots, \alpha_m$ by using equations (3) and (4)
2. Compute w_{eq} by using equation (6)

The algorithm is executed by P_0 when a new load needs processing. In order to compute $\boldsymbol{\alpha}$ and w_{eq}, processor P_i ($i = 1, \ldots m$) reports w_i to P_0. It is assumed that $z_1, z_2, \ldots, z_m$ are known to P_0.

We now consider the problem of scheduling divisible loads in *multi-level tree networks* [6]. Figure 3 depicts an execution on a multi-level tree distributed system composed of eight processors. Notice that after non-terminal processors receive load, they simultaneously begin computing a portion and transmitting the remainder of it to its children. A tree network comprises processor set $(\bar{P}_0, \ldots, \bar{P}_p)$ (with processing capacities $w_0, \ldots, w_p$ and link capacities $z_1, \ldots, z_p$), where $\bar{P}_0$ is the root. In the rest of the paper we denote by $\mathcal{S}_i$ the single-level subtree with root $\bar{P}_i$. The load is distributed from top to bottom, passing through each level. The TREE-LINEAR scheduling is defined similarly as SINGLE-LEVEL TREE-LINEAR where we desire to optimally distribute load across the tree such that we minimize the total execution time. The following theorem characterizes the optimal allocations for trees.

Theorem 2.3. *(Reduction)* The optimal solution is obtained by traversing tree $\mathcal{T}$ from bottom to top, replacing single-level subtrees with single equivalent processors until $\mathcal{T}$ is reduced to one processor. For single-level subtree $\mathcal{S}_j$, we compute load distribution $\boldsymbol{\alpha}^j$ and w_{eq}^j by SINGLE-LEVEL TREE-LINEAR algorithm and replace $\mathcal{S}_j$ with a processor with equivalent processing capacity w_{eq}^j.

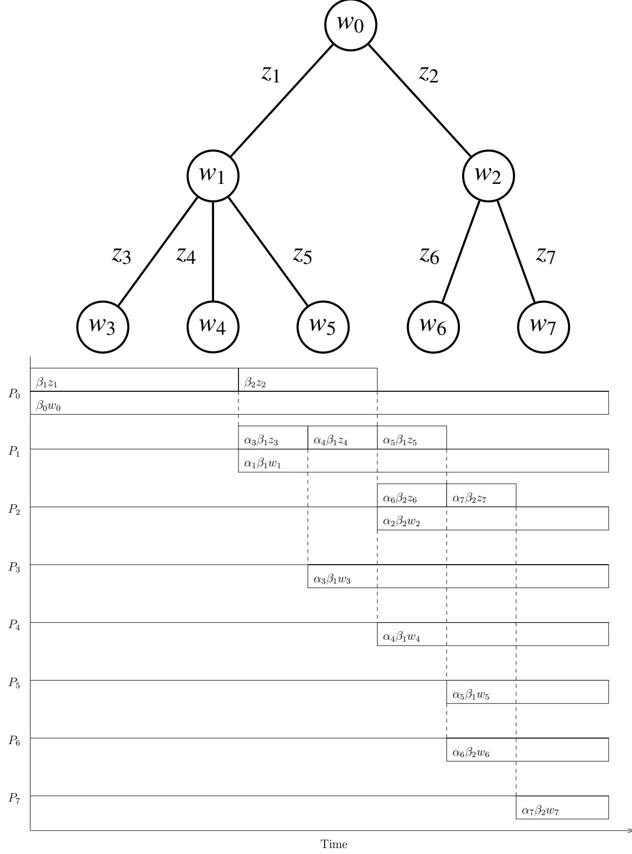

Figure 3. Execution on a 8-processor tree network

Proof. (Sketch) SINGLE-LEVEL TREE-LINEAR algorithm computes the optimal load distribution in a single-level tree and reduces it to a single processor with processing capacity w_{eq}. Assume that we have a single-level tree $\mathcal{S}_i$. Let tree $\mathcal{S}_i$ comprise processor set $(P_0^i, \ldots, P_{m_i}^i)$ (with link capacities $z_1^i \leq \ldots \leq z_{m_i}^i$), where $P_0^i = \bar{P}_i$. Let processor P_j^i $(j = 1, \ldots, m_i)$ be an equivalent processor that was defined using SINGLE-LEVEL TREE-LINEAR and thus is optimal. SINGLE-LEVEL TREE-LINEAR algorithm computes the optimal load distribution for processors $(P_0^i, \ldots, P_{m_i}^i)$. Since the load distribution for $\mathcal{S}_i$ is optimal and that the load distribution for the processors composing the subtree associated with equivalent processor P_j^i is optimal, then the load distribution for the processors in the subtree with root $\bar{P}_i$ is optimal. □

Theorem 2.4. *(Tree Participation)* The optimal solution for the TREE-LINEAR problem is obtained when all processors participate and they all finish executing their assigned load at the same time.

Proof. (Sketch) The theorem follows directly from reduction (Theorem 2.3). The SINGLE-LEVEL TREE-LINEAR algorithm results in all processors participating. Therefore, all processors in the multilevel tree network will participate when solving the TREE-LINEAR problem. For proving that all processors finish computing at the same time, assume that we have a single-level tree comprising of processor set $(P_0^j, \ldots, P_{m_j}^j)$ and that we have computed the optimal load distribution by the SINGLE-LEVEL TREE-LINEAR algorithm. All the processors finish at time T_j. Let processor P_i^j $(i = 1, \ldots, m_j)$ be an equivalent processor with root $\bar{P}_i$. Since equivalent processor P_i^j completes at time T_j, the processors in single-level tree $\mathcal{S}_i$ (the subtree rooted at $\bar{P}_i$) must complete at time T_j. We continue until the reductions are completely undone, thus proving that all processors in the subtree with root $\bar{P}_j$ stop at time T_j. □

The following algorithm solves the TREE-LINEAR problem.

Algorithm 2.2. (*TREE-LINEAR*)

Input: Processor capacities $w_0, \ldots, w_p$;
Link capacities $z_1, \ldots, z_p$;
Output: Load allocations $(\alpha_0^0, \ldots, \alpha_{m_0}^0), \ldots, (\alpha_0^p, \ldots, \alpha_{m_p}^p)$,
where $(\alpha_0^i, \ldots, \alpha_{m_i}^i)$ is the allocation for single-level tree $\mathcal{S}_i$;

1. **for** $i = 0, \ldots, p$; **do**
2. $\quad \alpha_0^i \leftarrow 1$
3. **until** a single processor remains; **do**
4. $\quad$ Find a single-level subtree $\mathcal{S}_i = (P_0^i, \ldots, P_{m_i}^i)$, where $P_0^i = \bar{P}_i$
5. $\quad \alpha_0^i, \ldots, \alpha_{m_i}^i; w_{eq}^i \leftarrow$SINGLE-LEVEL TREE-LINEAR$(w_0^i, \ldots, w_{m_i}^i; z_1^i, \ldots, z_{m_i}^i)$
6. $\quad$ Replace $\mathcal{S}_i$ with a single processor with processor capacity w_{eq}^i

In the above algorithm we set $\alpha_0^j = 1$ for terminal processor $\bar{P}_j$, which conveys the fact that $\bar{P}_j$ computes all the work that it receives. Note that $\alpha_0^j, \ldots, \alpha_{m_j}^j$ are the load allocation fractions for single-level tree $\mathcal{S}_j$ and not for the entire tree $\mathcal{T}$. The root P_0^j distributes load to processor P_i^j $(i = 1, \ldots, m_j)$ according to these fractions. Let $\bar{P}_i$ be the root of the subtree which corresponds to equivalent processor P_i^j. Processor $\bar{P}_j$ receives γ_j fractions of the total work and sends α_i^j fractions to $\bar{P}_i$. Processor $\bar{P}_i$ receives γ_i fractions of the total work given by

$$\gamma_i = \begin{cases} 1 & \text{if } i = 0, \\ \gamma_j \alpha_i^j & \text{if } i \neq 0. \end{cases} \tag{7}$$

At the start, the root processor $\bar{P}_0$ is the only processor with access to load and thus, $\gamma_0 = 1$. For any other processor $\bar{P}_i$ $(i \neq 0)$, its load is dependent on the load received by

its parent. Out of γ_i fractions of total work, processor $\bar{P}_i$ computes α_0^i fractions of it; thus, it computes $\alpha_0^i\gamma_i$ fractions of the total work.

In the above algorithm it is assumed that $\bar{P}_i$ reports its true processing capacity to its parent. 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 mechanism design for one parameter agents. Each agent in this mechanism design problem is characterized by private data represented by a single real value [20]. We define the problem in the following.

A *mechanism design problem for one parameter agents* is characterized by

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

(ii) Each agent i $(i = 1,\dots,m)$ has a privately known value t_i called the *true value* and a publicly known parameter $\tilde{w}_i \geq t_i$ called the *actual value*. 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 execution 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* $\alpha(\mathbf{w}) = (\alpha_1(\mathbf{w}),\dots,\alpha_m(\mathbf{w}))$. The input to this function is the vector of agents' bids $\mathbf{w} = (w_1,\dots,w_m)$ and returns an output $\alpha \in L$. (ii) The *payment function* $Q(\mathbf{w},\tilde{\mathbf{w}}) = (Q_1(\mathbf{w},\tilde{\mathbf{w}}),\dots,Q_m(\mathbf{w},\tilde{\mathbf{w}}))$ 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 maximizes 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 the inputs and the algorithm. An agent will implement an algorithm different from what is specified if it is beneficial to do so.

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

4. The Proposed Mechanism

We propose the DLS-TL mechanism for scheduling divisible loads in tree networks. The system model comprises a tree $\mathcal{T}$ of $p+1$ processors, where the root $\bar{P}_0$ is obedient and processors $\bar{P}_1,\dots,\bar{P}_p$ are strategic. Processor $\bar{P}_0$ is special as it performs functions the ensure mechanism validity. We assume that the links are obedient and that the communication protocols are tamper-proof.
Notation. We use the following notations in this section.

- Tree $\mathcal{T}$ has processor set $(\bar{P}_0,\dots,\bar{P}_p)$; processor $\bar{P}_0$ is the root of $\mathcal{T}$. Tree $\mathcal{S}_j$ is a single-level subtree of the reduced $\mathcal{T}$ with root P_0^j (with processing capacity w_0^j) and leaves $(P_1^j,\dots,P_{m_j}^j)$ (with processing capacities $w_1^j,\dots,w_{m_j}^j$ and link capacities $z_1 \leq \dots \leq z_{m_j}$). Processor P_0^j is an alias for $\bar{P}_j$. Equivalent processor

P_i^j $(i = 1, \dots, m_j)$ corresponds to the subtree rooted at $\bar{P}_i$. Processor $\bar{P}_k$ is the parent to processor $\bar{P}_j$.

- Let SK_i be the private key of processor $\bar{P}_i$. The secure digital signature of message m under SK_i is $sig(m)$. The message $dsm_i(m) = (m, sig_i(m))$ is the digitally signed message m under SK_i.

The description of DLS-TL mechanism follows. 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 recognizes its parent, grandparent, siblings, and children and is capable of verifying their signatures.

A processor $\bar{P}_i$ may process its assigned load at a different processing rate given by the *actual processing capacity*, where $\tilde{w}_i \geq t_i$. Thus, processor $\bar{P}_i$ may process its assigned load at a slower rate than its true processing capacity. 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 assigned loads using their full processing capacity. Each processor $\bar{P}_i$ is augmented with a tamper-proof meter that records $\tilde{w}_i$ and reports the value as $dsm_0(\tilde{w}_i)$.

DLS-TL Mechanism

Phase I (*Computing load allocation - bottom-up*) We apply the TREE-LINEAR algorithm with the following modifications to bid transmission. In single-level subtree $\mathcal{S}_j$, processor P_i^j transmits its digitally-signed bid $dsm_i(w_i^j)$ to parent P_0^j. We denote by $\mathbf{w}^j = (w_0^j, \dots, w_{m_j}^j)$ the vector of bids and by $\mathbf{w}_{dsm}^j$ the vector of signed bids. Processor P_0^j verifies the authenticity of the received messages and it will terminate the protocol if it receives contradictory messages from source P_i^j. Processor P_0^j submits the evidence to $\bar{P}_0$, which attempts to substantiate the claim. If the claim is true, P_i^j is fined C and P_0^j is rewarded C. If the claim is false, P_0^j is fined C and P_i^j is rewarded C. Fine C must be larger than any potential profits that are attainable by cheating. If P_i^j receives an unverifiable message, the protocol is terminated and no rewards or fines are issued as either sender or recipient may have invalidated the message. We denote by $\boldsymbol{\alpha}^j = (\alpha_0^j, \dots, \alpha_{m_j}^j)$ the vector of load fractions for subtree $\mathcal{S}_j$. Processor P_0^j computes $\boldsymbol{\alpha}^j$ and w_{eq}^j using the SINGLE-LEVEL TREE-LINEAR algorithm, and it then transmits $dsm_j(w_{eq}^j)$ to its parent $\bar{P}_k$. The correctness of the w_{eq}^j will be verified in Phase II. All load allocations $\boldsymbol{\alpha}^0, \dots, \boldsymbol{\alpha}^p$ are computed in this phase.

Phase II (*Computing load allocation - top-down*) We compute the allocation γ_i, which is the fraction of the total load that processor $\bar{P}_i$ receives. We recursively undue the transformations performed in Phase I. The root P_0^j sends message

$$G_i = (dsm_k(\gamma_j), dsm_j(\gamma_i), dsm_k(w_{eq}^j), dsm_j(w_0^j, \dots, w_m^j)) \tag{8}$$

to child P_i^j, where $\bar{P}_k$ is the parent of $\bar{P}_j$, γ_j is the fraction of total load transmitted to $\bar{P}_j$ from $\bar{P}_k$, and γ_i is the fraction of total load transmitted to $\bar{P}_i$ from $\bar{P}_j$ as computed by (7). Processor P_i^j verifies the various signatures and computations in G_i. Specifically, it verifies that $(w_0^j, \dots, w_m^j)$ results in w_{eq}^j and that $\gamma_i = \alpha_i^j \gamma_j$. Processor $\bar{P}_i$ terminates the protocol if it cannot validate the computations, receives contradictory messages, or cannot validate the signatures. In the first two scenarios, $\bar{P}_i$ submits a complaint to $\bar{P}_0$. If the complaint is confirmed, $\bar{P}_j$ is fined C and $\bar{P}_i$ is rewarded C. If the complaint is debunked, $\bar{P}_i$ is fined C and $\bar{P}_j$ is rewarded C. If a signature is unverifiable, all parities experience zero utility.

Phase III (*Load distribution and execution*) The load is distributed to the processors from root to leaves. Beginning with processor $\bar{P}_0$, processor P_0^j distributes α_i^j fractions of load to P_i^j; processor P_0^j computes α_0^j fractions itself. A processor begins computing its load as soon as it receives its entire assignment. Processor P_0^j may attempt to increase its utility by decreasing its fraction ($\tilde{\alpha}_0^j < \alpha_0^j$) and increasing its children fractions ($\tilde{\alpha}_i^j > \alpha_i^j$) such that $\sum_{l=0}^{m} \tilde{\alpha}_l^j = 1$. In this scenario, $\tilde{\gamma}_i > \gamma_i$, where $\tilde{\gamma}_i$ is the actual fraction of total work transmitted to processor $\bar{P}_i$. At this point, processor $\bar{P}_i$ has no choice but to accept the greater load. When computing completes, processor $\bar{P}_i$ notifies $\bar{P}_0$ about $\tilde{\gamma}_i > \gamma_i$. We assume that the data is embedded with a device [1] L_i such that L_i permits processor $\bar{P}_i$ to prove it has received $\gamma_i \leq \tilde{\gamma}_i$ fractions of load, *i.e.*, a processor cannot show more work than it has received. Processor $\bar{P}_0$ substantiates the claim by verifying that $\tilde{\gamma}_i > \gamma_i$ and, if the claim is false, $\bar{P}_i$ is fined C. If the claim is confirmed, processor $\bar{P}_0$ begins traversing the predecessors of $\bar{P}_i$. Let processors $\bar{P}_x$ and $\bar{P}_y$ be predecessors of $\bar{P}_i$ such that $\bar{P}_x$ is a child of $\bar{P}_y$; furthermore,

[1] An example of a simple device L_i would be preparing the data by dividing it into equal-sized blocks and appending a unique, random identifier to each block. The identifier space must be large enough so that the probability of an agent successfully guessing a valid identifier is small. Submitting the identifiers would permit $\bar{P}_i$ to show the amount of data it received.

let $\bar{P}_x$ be the root of equivalent processor P_x^y. Processor $\bar{P}_0$ fines C to all processors $\bar{P}_y$ where $\tilde{\gamma}_x > \alpha_x^y \tilde{\gamma}_y$. The fines are pooled and rewarded to $\bar{P}_i$. A processor may be fined multiple times depending on the number of processors impacted by its transgression.

Phase IV (*Payment computation*) The payment for processor $\bar{P}_i$ (with parent $\bar{P}_j$) is computed by $\bar{P}_i$. In the following, $\boldsymbol{\alpha}(\mathbf{w}^i) = (\alpha_0^i, \ldots, \alpha_{m_i}^i)$ is the allocations for single-level subtree $\mathcal{S}_i$ computed from bids $\mathbf{w}^i = (w_0^i, \ldots, w_{m_i}^i)$; $\mathbf{w}^j = (w_0^j, \ldots, w_{m_j}^j)$ are the bids for $\mathcal{S}_j$; $\tilde{\alpha}_0^i$ is the actual load allocation for $\bar{P}_i$; $\tilde{w}_i$ is the actual processing capacity for $\bar{P}_i$; and $\tilde{\gamma}_i$ is the actual fraction of total work received by $\bar{P}_i$. The goal of processor $\bar{P}_i$ is to maximize its utility. The *utility* of $\bar{P}_i$ is

$$U_i(\boldsymbol{\alpha}(\mathbf{w}^i), \mathbf{w}^j, \tilde{\alpha}_0^i, \tilde{w}_i, \tilde{\gamma}_i) = V_i(\tilde{\alpha}_0^i, \tilde{w}_i, \tilde{\gamma}_i) + Q_i(\boldsymbol{\alpha}(\mathbf{w}^i), \mathbf{w}^j, \tilde{w}_i, \tilde{\gamma}_i), \quad (9)$$

where V_i is the valuation function and Q_i is the payment function. The *valuation function* V_i is

$$V_i(\tilde{\alpha}_0^i, \tilde{w}_i, \tilde{\gamma}_i) = -\tilde{\gamma}_i \tilde{\alpha}_0^i \tilde{w}_i \quad (10)$$

The *payment function* Q_i is

$$Q_i(\boldsymbol{\alpha}(\mathbf{w}^i), \mathbf{w}^j, \tilde{w}_i, \tilde{\gamma}_i) = \begin{cases} 0 & \text{if } \tilde{\gamma}_i = 0, \\ C_i(\boldsymbol{\alpha}(\mathbf{w}^i), \tilde{w}_i, \tilde{\gamma}_i) + B_i(\boldsymbol{\alpha}(\mathbf{w}^i), \mathbf{w}^j, \tilde{w}_i) & \text{if } \tilde{\gamma}_i > 0, \end{cases} \quad (11)$$

where

$$C_i(\boldsymbol{\alpha}(\mathbf{w}^i), \tilde{w}_i, \tilde{\gamma}_i) = \tilde{\gamma}_i \alpha_0^i \tilde{w}_i \quad (12)$$

is the *compensation function* and

$$B_i(\boldsymbol{\alpha}(\mathbf{w}^i), \mathbf{w}^j, \tilde{w}_i) = w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - w_{eq}^j(\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, w_i^j)), (\mathbf{w}_{-i}^j, \hat{w}_i)) \quad (13)$$

is the *work bonus function*. The function $w_{eq}^j(\alpha(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j)$ is the optimal equivalent processing capacity of single-level subtree $\mathcal{S}_j$ without subtree $\mathcal{S}_i$. The function $w_{eq}^j(\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, w_i^j)), (\mathbf{w}_{-i}^j, \hat{w}_i))$ is the equivalent processing capacity adjusted for the actual processing capacity of $\bar{P}_i$, where

$$\hat{w}_i = \begin{cases} \tilde{w}_i & \text{if } \bar{P}_i \text{ is a terminal processor in tree } \mathcal{T}, \\ \tilde{w}_i \alpha_0^i & \text{if } \tilde{w}_i \geq w_i, \\ w_i^j & \text{if } \tilde{w}_i < w_i. \end{cases} \quad (14)$$

In the above, we adjust the performance of subtree $\mathcal{S}_i$ for the actual performance of processor $\bar{P}_i$. In the event that $\bar{P}_i$ runs slower ($\tilde{w}_i > w_i$), the equivalent processing capacity of $\mathcal{S}_i$, w_{eq}^i, is dominated by $\bar{P}_i$'s performance; if $\bar{P}_i$ runs faster ($\tilde{w}_i < w_i$), w_{eq}^i remains unchanged from w_i^j. Processor $\bar{P}_i$ saves

$$Proof_i = (G_i, \mathbf{w}_{dsm}^i, dsm_0(\tilde{w}_i), dsm_i(\tilde{\gamma}_i), L_i) \quad (15)$$

as evidence of correctly computing the payment. It submits bill Q_i to the payment infrastructure. With probability q (where $0 < q \leq 1$), processor $\bar{P}_0$ requests $Proof_i$ from $\bar{P}_i$. If P_i fails to provide a valid proof for Q_i, it receives fine C/q.

This concludes the description of the DLS-TL mechanism. The mechanism as described above is valid for selfish-but-agreeable agents but not 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 algorithm if it is the only action that maximizes its welfare. In the case of DLS-TL, 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 (11) to

$$Q_i(\boldsymbol{\alpha}(\mathbf{w}^{\mathbf{i}}), \mathbf{w}^j, \tilde{w}_i, \tilde{\gamma}_i) = \begin{cases} 0 & \text{if } \tilde{\gamma}_i = 0, \\ C_i(\boldsymbol{\alpha}(\mathbf{w}^i), \tilde{w}_i, \tilde{\gamma}_i) + B_i(\boldsymbol{\alpha}(\mathbf{w}^i), \mathbf{w}^j, \tilde{w}_i) + S & \text{if } \tilde{\gamma}_i > 0, \end{cases} \quad (16)$$

where $S = 0$ if a solution is not found, and s if a solution is found. The function S is called the *solution bonus*. 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.

5. DLS-TL Properties

In this section we study the properties of the DLS-TL mechanism. The first property we investigate is strategyproofness. To prove this, we first show that all the processors do not have incentives to deviate from the algorithm. Then we show that if the processors do not deviate (but they can still execute differently than they bid), then truthful processors receive the maximum profit. Combining the two, we prove that the mechanism is strategyproof.

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

Proof. Let processor $\bar{P}_i$ be a selfish-but-agreeable agent and $\bar{P}_h$ be a child of $\bar{P}_i$. A selfish-but-agreeable agent will deviate if the action increases its utility. Process $\bar{P}_i$ may deviate from the protocol by either (i) sending contradictory messages, (ii) incorrectly computing w_{eq}^i or γ_h, (iii) decreasing its workload by increasing the workload of its children (*i.e.*, $\tilde{\gamma}_h > \gamma_h$), (iv) requesting a payment greater than Q_i, (v) falsely accusing a processor of cheating. Processor $\bar{P}_i$ will not deviate in other fashions (*e.g.*, corrupt data) because there is no benefit to do so. To combat deviation, incentives are provided to processors for monitoring one another. In case (i), the recipient will report $\bar{P}_i$ and obtain reward C; processor $\bar{P}_i$ will be fined C. Fine C is a deterrent as it is greater than any profit attainable by cheating. In case (ii), child $\bar{P}_h$ will verify the computations in Phase II and will report $\bar{P}_i$ for reward C if the computation cannot be validated. Again, $\bar{P}_i$ is fined C for deviating from the algorithm. In case (iii), child $\bar{P}_h$ receives a reward of at least C for reporting $\bar{P}_i$ and processor $\bar{P}_i$ is penalized C. In case (iv), the fine C/q (where $0 < q \leq 1$ is the probability of processor $\bar{P}_0$ requesting a proof) is a deterrent for over billing. The complete proof for case (iv) can be found in [16]. In case (v), processor $\bar{P}_i$ does not have the evidence to support its claim. Thus, it will be fined C. □

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

Proof. Processor $\bar{P}_i$ is fined for either deviating from the protocol or another processor $\bar{P}_j$ produces contradictory messages signed by $\bar{P}_i$. In the first case, $\bar{P}_i$ clearly deviates from DLS-TL. In the second case, $\bar{P}_j$ sends the messages either by successfully forging signatures or by possessing the private key SK_i. We assume that the forging of signatures is impossible. Processor $\bar{P}_j$ obtains SK_i either by $\bar{P}_i$ sharing it or by stealing it from $\bar{P}_i$. It is a violation of the mechanism for a second party to possess SK_i. Thus, $\bar{P}_i$ 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-TL.

Proof. From Lemma 5.1 and 5.2, a selfish-but-agreeable processor will be fined for and only for deviating. Therefore, the processor does not have incentives to deviate from DLS-TL. □

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

Proof. Let processor $\bar{P}_i$ be a selfish-and-annoying agent. Theorem 5.1 handles the cases in which deviation results in greater utility, *i.e.*, $U_i' - U_i > 0$, where U_i' is the utility of a deviating $\bar{P}_i$. Processor $\bar{P}_i$ will also deviate if it does not reduce its utility, *i.e.*, $U_i' - U_i = 0$. Such actions include data corruption and sending the same data to different children. These actions reduce the possibility of obtaining a solution to the given problem and thus, reduce the possibility of receiving the solution bonus. Processor $\bar{P}_i$ is welfare maximizing and thus, will not choose to do such. Therefore, processor $\bar{P}_i$ does not have incentives to deviate from DLS-TL. □

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

Proof. Assume processor $\bar{P}_j$ is the parent of $\bar{P}_i$. The utility U_i of processor $\bar{P}_i$ $(i = 1, \ldots, p)$ is

$$\begin{aligned} U_i &= V_i + Q_i \\ &= -\tilde{\gamma}_i \tilde{\alpha}_0^i \tilde{w}_i + \tilde{\gamma}_i \alpha_0^i \tilde{w}_i + \\ &\quad w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - w_{eq}^j(\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, w_i^j)), (\mathbf{w}_{-i}, \hat{w}_i)) \end{aligned}$$

We assume that the processors do not deviate and thus abide by the computed load allocations; therefore, $\tilde{\alpha}_0^i = \alpha_0^i$. The utility U_i is

$$U_i = w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - w_{eq}^j(\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, w_i^j)), (\mathbf{w}_{-i}, \hat{w}_i)). \tag{17}$$

We consider two cases:

(i) $\tilde{w}_i = t_i$, *i.e.*, processor $\bar{P}_i$ computes the load at full capacity. Assume $\bar{P}_i$ to be a terminal processor in tree $\mathcal{T}$. If $\bar{P}_i$ bids its true value $(w_i^{[e]} = t_i)$, then its utility $U_i^{[e]}$ is

$$\begin{aligned} U_i^{[e]} &= w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - w_{eq}^j((\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, t_i)), (\mathbf{w}_{-i}^j, t_i)) \\ &= w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - w_{eq}^{j[e]}. \end{aligned} \tag{18}$$

If $\bar{P}_i$ bids lower $(w_i^{[l]} < t_i)$, then its utility $U_i^{[l]}$ is

$$\begin{aligned} U_i^{[l]} &= w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}) - w_{eq}^j(\boldsymbol{\alpha}((\mathbf{w}_{-i}, w_i^{[l]})), (\mathbf{w}_{-i}, t_i)) \\ &= w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}), \mathbf{w}_{-i}) - w_{eq}^{j[l]}. \end{aligned} \tag{19}$$

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

$$\begin{aligned} U_i^{[h]} &= w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - w_{eq}^j(\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, w_i^{[h]})), (\mathbf{w}_{-i}^j, t_i)) \\ &= w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - w_{eq}^{j[h]}. \end{aligned} \tag{20}$$

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 $\bar{P}_i$ and increased load to the other processors. Since $\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, t_i))$ is optimal, $w_{eq}^{j[e]} \leq w_{eq}^{j[h]}$ and thus, $U_i^{[e]} \geq U_i^{[l]}$.

We now assume $\bar{P}_i$ to be an interior processor of tree $\mathcal{T}$. If $\bar{P}_i$ bids its true value ($w_i^{[e]} = t_i$), then its utility $U_i^{[e]}$ is

$$\begin{aligned} U_i^{[e]} &= w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - \\ &\quad w_{eq}^j(\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, w_i^{j[e]})), (\mathbf{w}_{-i}, w_i^{j[e]})) \\ &= w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - \\ &\quad w_{eq}^j(\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, w_i^{j[e]})), (\mathbf{w}_{-i}^j, \alpha_0^{i[e]} t_i)) \qquad \text{by (5)} \\ &= w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - w_{eq}^{j[e]}, \end{aligned} \tag{21}$$

where $w_i^{j[e]}$ is the processing capacity of $\mathcal{S}_i$. If $\bar{P}_i$ bids lower ($w_i^{[l]} < t_i$), then its utility $U_i^{[l]}$ is

$$\begin{aligned} U_i^{[l]} &= w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}), \mathbf{w}_{-i}^j) - \\ &\quad w_{eq}^j(\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, w_i^{j[l]})), (\mathbf{w}_{-i}^j, \alpha_0^{i[l]} w_i^{[l]})) \\ &= w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - w_{eq}^{j[l]}, \end{aligned} \tag{22}$$

where $w_i^{j[l]}$ is the processing capacity of $\mathcal{S}_i$ and $\alpha_0^{i[l]}$ is the fraction of load assigned to $\bar{P}_i$, both of which are computed with bids $(w_i^{[l]}, w_1^i, \ldots, w_{m_i}^i)$. We know that $\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, w_i^{j[e]}))$ is the optimal allocation by the SINGLE-LEVEL TREE-LINEAR algorithm. By bidding lower, $\bar{P}_i$ is assigned more load, *i.e.*, $\alpha_0^{i[l]} \geq \alpha_0^{i[e]}$. The performance of the single-level subtree is constrained by $\bar{P}_i$. Thus, $w_{eq}^{j[e]} \leq w_{eq}^{j[l]}$ which proves $U_i^{[e]} \geq U_i^{[l]}$. Finally, if $\bar{P}_i$ bids higher ($w_i^{[h]} \geq t_i$), then its utility $U_i^{[h]}$ is

$$\begin{aligned} U_i^{[h]} &= w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - w_{eq}^j(\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, w_i^{j[h]})), (\mathbf{w}_{-i}^j, w_i^{j[h]})) \\ &= w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - w_{eq}^{j[h]}. \end{aligned} \tag{23}$$

where $w_i^{j[h]} = \alpha_0^{j[h]} w_i^{[h]}$. We know that $\alpha((\mathbf{w}_{-i}, w_i^{j[e]}))$ is the optimal allocation. By bidding higher, greater load is assigned to the other processors which reduces the performance of subtree $\mathcal{S}_j$. This results in $w_{eq}^{[e]} \leq w_{eq}^{[h]}$; 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-TL mechanism is strategyproof.

Proof. Lemma 5.3 states that if no deviation occurs, the mechanism is strategyproof. Theorems 5.1 and 5.2 state that processors have no incentives to deviate from the algorithm. Therefore, the mechanism is strategyproof. □

Another useful property is voluntary participation. When a mechanism satisfies the voluntary participation condition, a truthful processor will never obtain negative utility (*i.e.*, $U_i \geq 0$).

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

Proof. The utility of processor $\bar{P}_i$ (with parent $\bar{P}_j$) when it bids its true value is

$$U_i = w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - w_{eq}^j(\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, w_i^j)), (\mathbf{w}_{-i}^j, \hat{w}_i)). \tag{24}$$

If $\bar{P}_i$ is an intermediate processor, then $w_i^j = \hat{w}_i = \alpha_0^i t_i$. If it is a terminal processor, then $\hat{w}_i = t_i$. If we let $\alpha_0^i = 1$ for terminal processor $\bar{P}_i$ (as we did in Section 2), then we simplify (24) to

$$U_i = w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j) - w_{eq}^j(\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, \alpha_0^i t_i)), (\mathbf{w}_{-i}^j, \alpha_0^i t_i)). \tag{25}$$

The equivalent processing capacity $w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j)$ is obtained by using all the processors except for $\bar{P}_i$ and its subtree $\mathcal{S}_i$. By Theorem 2.4, we know that the optimal equivalent processing capacity w_{eq}^j is obtained when all processors participate and thus, $w_{eq}^j(\boldsymbol{\alpha}((\mathbf{w}_{-i}^j, \alpha_0^i t_i)), (\mathbf{w}_{-i}^j, \alpha_0^i t_i)), (\mathbf{w}_{-i}^j, \alpha_0^i t_i)) \leq w_{eq}^j(\boldsymbol{\alpha}(\mathbf{w}_{-i}^j), \mathbf{w}_{-i}^j)$. Therefore, the utility is $U_i \geq 0$. □

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

Proof. We construct the proof similar to Theorem 5.3. Lemma 5.4 states that the mechanism satisfies voluntary participation if no deviation occurs. By Theorems 5.1 and 5.2, we know that processors will not deviate. Hence, DLS-TL mechanism satisfies the voluntary participation condition. □

Due to the lack of space, we quickly examine the communication complexity of DLS-TL. Disregarding the communication with $\bar{P}_0$ for algorithm enforcement and the distribution of load, we see that Phase II dominates communication in terms of message quantity and size. In that phase, the root of a single-level subtree must send a $m+1$-sized message to each of its m children. The worst case is then a single-level tree with n processors in which the root must transmit an n-sized message to each of its $n-1$ children; thus, the communication complexity in the worst-case is $\Theta(n^2)$.

6. Conclusion

In this paper we proposed the strategyproof mechanism DLS-TL for scheduling divisible loads in tree networks. In a tree network, the load originates at the root and it is dispersed through intermediate processors until the load is distributed to all. Through the use of incentives, processors report and process their assignments at full capacity. Incentives are also provided for reporting algorithm deviation. A processor will readily report a deviant in order to receive a reward. The fine for deviation is greater than any profits attainable by cheating, which will dissuade processors form attempting it. In a final note, DLS-TL satisfies the voluntary participation condition. All truthful, non-deviating processors will obtain non-negative utility.

We are planning to continue our investigation into the augmentation of DLT with incentives. By examining various influences such as network architectures and the rational behavior of links, we hope to achieve a cohesive theory combining DLT with incentives.

Acknowledgment. This research was supported in part by Wayne State University Faculty Research Grant.

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] A. Archer and E. Tardos. Frugal path mechanisms. In *Proc. of the 13th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 991–999, Jan. 2002.

[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] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] C. Ng, D. Parkes, and M. Seltzer. Strategyproof computing: Systems infrastructures for self-interested parties. In *Proc. of the 1st Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[18] C. Ng, D. Parkes, and M. Seltzer. Virtual worlds: Fast and strategyproof auctions for dynamic resource allocation. In *Proc. of the ACM Conference on Electronic Commerce*, pages 238–239, June 2003.

[19] N. Nisan, S. London, O. Regev, and N. Camiel. Globally distributed computation over Internet - The POPCORN project. In *Proc. of the 18th IEEE International Conference on Distributed Computing Systems*, pages 592–601, May 1998.

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

[21] M. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, Cambridge, Mass., 1994.

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

[23] 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.

[24] R. Wolski, J. S. Plank, T. Bryan, and J. Brevik. G-commerce: market formulations controlling resource allocation on the computational grid. In *Proc. of the 15th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2001.

[25] D. Yu and T. G. Robertazzi. Divisible load scheduling for grid computing. In *Proc. of the 15th International Conference on Parallel and Distributed Computing and Systems*, pages 1–8, Nov. 2003.