Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT•

Computer Communications 27 (2004) 93–110

computer communications

www.elsevier.com/locate/comcom

Divisible load scheduling strategies on distributed multi-level tree networks with communication delays and buffer constraints

Bharadwaj Veeravalli*, Jingnan Yao

Open Source Software Laboratory (http://opensource.nus.edu.sg), Department of Electrical and Computer Engineering, The National University of Singapore, 4 Engineering Drive 3, Singapore, Singapore 117576

Received 22 August 2002; revised 19 May 2003; accepted 20 May 2003

Abstract

In this paper, the problem of scheduling divisible loads on arbitrary tree networks is considered, with the objective to minimize the total processing time of the entire load. We consider an arbitrary tree network comprising heterogeneous processors interconnected via heterogeneous links. The divisible load is assumed to originate at the root processor in the network and the processors in the system are assumed to render only a finite amount of buffer space (capacity constraint) for processing the divisible load. For a special case of the problem instance wherein when all the processor-link pairs in the tree network can be utilized, we design an algorithm referred to as *pull–push optimal load distribution* algorithm to obtain an optimal solution. For the generic case, when the network has any redundant processor-link pairs (those pairs whose consideration in scheduling the load will penalize the performance) we propose three heuristic strategies to obtain the best possible time performance. These heuristics are based on applying systematic procedures to eliminate any redundant-link pairs and also using the optimal sequencing theorem proposed in the literature for single-level tree networks. We derive the worst and the best case complexities for all the algorithms proposed and present a discussion on the applicability of the algorithms on a wide range of network scenarios. We demonstrate all the above strategies with illustrative examples and evaluate their performance.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Load distribution; Tree networks; Divisible loads; Processing time; Communication delays; Capacity constraints

1. Introduction

Network-based computing has been proven to be a powerful tool in scheduling and processing computationally intensive loads for various applications in the past decade. Applications such as large scale simulation in computational fluid-dynamics, remote control/sensor systems, solutions to N-body problems, edge-detection applications in image processing, implementing numerical algorithms for large scale data, etc. [1–3] are few such examples. Handling such large volume of loads demands a clever design of polices to partition, communicate, and to process the load on a network of computers. The theory of scheduling and processing of divisible loads, referred to as *divisible load theory* (DLT), since its origin in 1988 [4], has stimulated considerable interest among researchers in the field of parallel and distributed systems. Divisible loads belong to a class of loads that demand CPU intensive computations and are usually very large in size, homogeneous, and are arbitrarily divisible.

The load being assumed to be arbitrarily divisible, each partitioned portion of the load can be independently processed on any processor on the network and each portion demands identical processing requirements (homogeneous). DLT adopts a linear mathematical modeling of the processor speed and communication link speed parameters. In this model, the communication time delay is assumed to be proportional to the amount of load that is transferred over the channel, and the computation time is proportional to the amount of load assigned to the processor. The communication model assumed is predominantly a *uniport* model, by which we mean that the source processor communicates the load fractions to a set of destination processors one at a time. The primary objective in the research of DLT is to determine the *optimal fractions* (distribution) of the entire load to be assigned to each of the processors such that the total processing time of the entire load is a minimum.

* Corresponding author. Tel.: +65-68745158; fax: +65-67791103.
E-mail addresses: elebv@nus.edu.sg, engp0969@nus.edu.sg (B. Veeravalli).

Compilation of all the research contributions in DLT until 1996 can be found in the monographs [5,6] and a recent report summarizes all the published contributions till date (2000) in this domain [7]. Therefore, we shall now present a very brief survey on some of the significant contributions in this area relevant to the problem addressed in this paper and the readers are referred to Ref. [7] for an up-to-date survey.

In all the research so far in this domain, a criterion that is used to derive optimal solution is as follows. It states that in order to obtain an optimal processing time, it is *necessary and sufficient* that all the processors participating in the computation must stop computing at the same time instant. This condition is referred to as an *optimality principle* in the DLT literature and analytic proof of the assumption for optimal load distribution for bus networks also appears in Ref. [8]. The effect of topologies on load distribution was presented in Ref. [9] and in this study, the load distribution problem was analyzed on a variety of computer networks such as linear, mesh and hypercube. Barlas [10] presented an important result concerning an optimal sequencing in a tree network by including the results collection phase. Load partitioning of intensive computations of large matrix-vector products in a multicast bus network was theoretically investigated in Ref. [11].

A recent paper [12] introduced simultaneous use of communication links to expedite the communication and the concept of *fractal hypercube* on the basis of *processor isomorphism* to obtain the optimal solution with fewer processors is proposed. Research efforts after 1996 particularly started focusing on including practical issues such as, scheduling under real-time deadline constraints [1], scheduling multiple divisible loads [13,14], scheduling divisible loads with arbitrary processor release times in bus networks [15], use of affine models for communication and computation for scheduling divisible loads [16,17], scheduling with finite-size buffers [18]. Finally, some of the proposed algorithms were tested using experiments on real-life application problems such as image processing [19], database operations [20], matrix-vector product computations [11]. In Ref. [21] rigorous experimental implementation of the matrix-vector products on PC clusters as well as on a network of workstations was carried out. In Ref. [20], several other applications such as *pattern search*, *file compression*, *joining operation in relational databases*, *graph coloring and genetic search* using the divisible load paradigm, were implemented.

In this paper, we consider the problem of scheduling arbitrarily divisible loads on a multi-level unbalanced tree network. Depending on the communication and computation models, one may design a variety of strategies for distributing the load. In this paper, we shall follow the model adopted in DLT as explained above and consider scheduling on arbitrary tree networks. As an added constraint, we consider the problem under the situation wherein processors in the network have finite buffer capacities to render for processing divisible loads. Thus, every processor in our model can render only a finite amount of buffer space for processing a divisible load. The problem of load distribution under this constraint was first developed in the study for single-level tree networks and an algorithm referred to as *incremental balancing strategy* (IBS) was proposed in Ref. [18]. For a special case, we design an algorithm referred to as *pull–push optimal load distribution* (PPOLD) algorithm and demonstrate its performance. For a generic case, we design three different strategies for load distribution and demonstrate their workings.

The organization of the paper is as follows. In Section 2, we present the problem formulation and in Section 3, we first propose an algorithm PPOLD to obtain an optimal processing time when a condition, referred to as Rule A, introduced in the literature [5], is satisfied. For continuity purposes, Appendix A gives a brief note and important relationships to be used while applying Rule A. Further, we propose three heuristic strategies to tackle the case when Rule A is not satisfied. Illustrative examples for all these algorithms/strategies are given in this section as well. In Section 4, we present several important discussions and applicability of the algorithms to a wide range of network scenarios. Further, we analyze the worst and the best case complexities of all the algorithms proposed in detail. Finally, in Section 5, we conclude the paper with future possible extensions to the research contributions reported in this paper.

2. Problem formulation

In this section, we shall introduce the problem formally and present the required definitions, notations, and terminology that will be used throughout the paper. As mentioned in Section 1, we consider the problem of scheduling and processing a divisible load on a generic tree network, denoted as G_T, consisting of N processors interconnected via $(N-1)$ communication links with $Q \geq 1$ levels. The processors and the links are assumed to be heterogeneous in the sense that their respective speeds may not be identical. The processing load is assumed to originate at the root processor and that all the processors in the system have front-ends. This means that each processor can compute and communicate with another processor to which it is connected directly via a link, simultaneously. Further, each front-end cannot receive and transmit the loads simultaneously. In addition, each processor is assumed to have a finite buffer to process the data. Although a processor can render only a finite buffer space as its share for computation of the divisible load, we shall assume that a (parent) processor can continue to distribute the load to its child processors even if its buffer capacity is fully utilized at a particular iteration. Thus, a processor whose buffer is filled completely can continue to participate, as a router, in the load distribution process at any time, as front-ends are the primary units which are responsible for communication of the load between any two processors.

Without loss of generality, we shall assume that all the processors in the system are capable of processing the load. Thus, in this paper, as mentioned in Section 1, we are interested in obtaining an optimal load distribution, the distribution that yields a minimum processing time, on arbitrary tree networks. We consider a fixed sequence of load distribution, i.e. we follow a sequence in which the load distribution is from the left end to the right end of the tree starting from the root, for every single-level tree at each level. When the processors have infinite buffer capacities, the solution obtained by this procedure will be an optimal solution *if and only if* every sub-tree (we refer to a single-level tree network as a sub-tree, hereafter) in G_T has links that satisfy a condition referred to as *Rule A* in the literature [5]. Rule A proposes a methodology by which redundant processor-link pairs (slow processor-link pairs in a given fixed sequence of load distribution) are eliminated in a sub-tree and generates an *optimal reduced tree* network that yields an optimal solution.

2.1. Definitions, notations, and some terminology

Since we have buffer resource constraints, we attempt to use the basic idea proposed in the algorithm IBS [18] in computing the equivalent buffer availability at each processor at each iteration. Table 1 presents the necessary notations, definitions, and the terminology that are used throughout the paper.

In our algorithms, we will be predominantly using the concept of processor-link equivalence. The process of obtaining equivalent processor and link parameters [5] has been described in brief in Appendix A. Essentially, in the Appendix A, we refer to important relationships that will be used in our algorithms for generating the equivalent sub-trees. Readers are referred to Ref. [5] for further details.

3. Design of load distribution strategies

In this section, our focus is to compute an optimal distribution for G_T with finite-size buffers following a given (fixed) sequence. As mentioned earlier, we use IBS [18] algorithm recursively (together with equivalent processor-link/buffer concepts) to derive the amount of buffer availability at each iteration in all our proposed strategies.

Here, we briefly describe the algorithm IBS for the purpose of continuity. IBS was designed to determine the optimal load distribution in single-level tree networks (assuming $(m+1)$ processors) with finite-size buffers. The algorithm works in an incremental fashion by optimally scheduling the load among the processors in an incremental fashion in every iteration. In brief, the algorithm attempts to use the optimality criterion in every iteration and attempts to schedule an *optimal* portion of the load that can be accommodated by each processor in every iteration, upon a comparison with the optimal load distribution under infinite buffer case. This comparison guarantees that at the end of each iteration, at least one buffer is completely filled. Consequently, the entire load is scheduled among the processors in no less than $O(m+1)$ iterations.

For our current problem, we need to consider two distinct cases—(a) when Rule A is satisfied by all the processor-link pairs in the entire G_T, and (b) when Rule A is not satisfied by some of the processor-link pairs in the entire G_T. When we assume that Rule A is satisfied for all the processor-link pairs for a fixed sequence, as long as the total buffer capacity of all the processors in G_T is greater than L, we propose an algorithm referred to as PPOLD to obtain an optimal solution. On the other hand, considering the general situation without the assumption of Rule A, PPOLD may not generate an optimal solution. This can be realized immediately after observing the working style of PPOLD for case (b), later. Thus, in this case, we attempt several heuristic strategies to obtain a sub-optimal solution. In both cases, initially, we assume that the total buffer capacity of the processors which satisfy Rule A in G_T is strictly greater than L. While this assumption guarantees that the entire load can be processed even without redundant processors that are eliminated during the application of Rule A in G_T, for case (b) above, the final processing time solution obtained by using G_T alone *need not guarantee* an optimal solution. Thus, for case (b), we inadvertently attempt to exclude those redundant processor-link pairs in each iteration. However, this still may not be sufficient to obtain an optimal solution, and hence, we consider including the set of redundant processor-link pairs of G_T which were eliminated in the previous iterations, as and when required, in a systematic fashion. Moreover, a redundant processor-link pair will be included only when it does not violate Rule A in the current iteration. We will also demonstrate this aspect in Example 2.

Remarks: A subtle point to be noted at this stage is the following. Comparing the above two cases, we can observe that, in both cases, processors whose buffers are filled completely will not participate in the subsequent iterations. Whereas, in case (b), since we are applying Rule A in every iteration, the number of processors that are participating in an iteration may vary, as we attempt to include redundant processor-link pairs (those processors that can continue rendering buffer space) that were eliminated in the previous iterations. Consequently, it becomes important that we denote G_T as $G_T^{(k)}$, during the kth iteration.

3.1. Optimal load distribution in G_T: case (a)

Although we use the idea of incrementally allocating the load by following the optimality principle, in our current context, the application of IBS needs to be carefully examined. While we start with G_T at the first iteration, the number of processors that participate in the next iteration will be different due to an important

Table 1
Glossary of notations

L	Total amount of load originating at the root processor for processing
$p_{x,i}$	Processor i in the tree G_T, where x is the index of its parent processor. Note that for the root processor of G_T, we simply denote it as p_0
$l_{p_{x,i},p_{y,j}}$	Communication link connecting processors $p_{x,i}$ and $p_{y,j}$ in the tree G_T
$\Sigma(x,i,m+1)$	This is a single-level tree network (sub-tree) defined in G_T, consisting of $(m+1)$ processors, with root processor $p_{x,i}$ and m child processors denoted as $p_{i,1}$, $p_{i,j}$, $p_{i,m}$. Further, since every child processor has the same parent in this sub-tree, we can conveniently denote the communication link $l_{p_{x,i},p_{i,j}}, j \in (1,\ldots,m)$ connecting $p_{x,i}$ with $p_{i,j}$ simply as $l_{i,j}$. Note that for the single-level tree with root processor p_0, we denote it as $\Sigma(0,m+1)$. We refer to this single-level tree as $\Sigma^{(k)}(x,i,m+1)$ at iteration k
T_{cp}	Time taken to process a unit load by a standard processor
T_{cm}	Time taken to communicate a unit load on a standard link
$w_{x,i}$	A constant that is inversely proportional to the speed of processor $p_{x,i}$. Similarly, $w_{i,j}$ is the inverse of the *speed* of processor $p_{i,j}$ in $\Sigma(x,i,m+1)$
$z_{x,i}$	A constant that is inversely proportional to the speed of link $l_{x,i}$. Similarly, $z_{i,j}$ is the inverse of the *speed* of link $l_{i,j}$ in $\Sigma(x,i,m+1)$
$p^{(k)}_{x,\mathrm{eq}(i)}$	An equivalent processor of the entire network $\Sigma^{(k)}(x,i,m+1)$. That is, we replace the entire sub-tree rooted at processor i with its equivalent processor. Note that, processor x now becomes the parent of this equivalent processor. Hence, we denote the respective speed parameter (inverse of the speed) of this equivalent processor as $w^{(k)}_{x,\mathrm{eq}(i)}$, at the kth iteration. Similarly, the equivalent processor of $\Sigma^{(k)}(0,m+1)$ is denoted as $p^{(k)}_{\mathrm{eq}(0)}$
$l^{(k)}_{x,\mathrm{eq}(i)}$	An equivalent link, which is equivalent to the communication speeds of a set of links in $\Sigma^{(k)}(x,i,m+1)$. The respective speed parameter of this equivalent link is denoted by $z^{(k)}_{x,\mathrm{eq}(i)}$, at the kth iteration
$B_{x,i}$	The initial buffer capacity of $p_{x,i}$. Further, we denote $B^{(k)}_{x,i}$ as the available buffer capacity of $p_{x,i}$ at the start of the kth iteration. Initially, $B^{(0)}_{x,i} = B_{x,i}$. Similarly, we denote the available buffer capacity of processor $p_{i,j}$ in $\Sigma^{(k)}(x,i,m+1)$ at the kth iteration as $B^{(k)}_{i,j}$
$B^{(k)}_{x,\mathrm{eq}(i)}$	The equivalent buffer capacity of the equivalent processor $p^{(k)}_{x,\mathrm{eq}(i)}$ at the kth iteration. Similarly, the equivalent buffer capacity of $p^{(k)}_{\mathrm{eq}(0)}$ is denoted as $B^{(k)}_{\mathrm{eq}(0)}$
$Y^{(k)}_{x,i}$	An optimal amount of load that can be processed (or equivalently, an optimal buffer space that can be rendered) in $\Sigma^{(k)}(x,i,m+1)$ at the kth iteration
α	This is defined as a N-tuple and refers to a load distribution, i.e. $\alpha = (\alpha_0, \alpha_{0,1}, \ldots, \alpha_{0,m}, \ldots, \alpha_{x,i}, \ldots, \alpha_{g,1}, \alpha_{g,2}, \ldots, \alpha_{g,m})$, where $\alpha_{x,i}$ is the fraction of load assigned to $p_{x,i}$ such that $0 \le \alpha_{x,i} \le 1$ and sum of all the above load fractions amounts to the total load to be processed. We denote the load distribution generated in the kth iteration as $\alpha^{(k)}$
$\alpha^{(k)}_{x,\mathrm{eq}(i)}$	Load fraction given to the equivalent processor $p^{(k)}_{x,\mathrm{eq}(i)}$ for processing. Note that for $p^{(k)}_{\mathrm{eq}(0)}$ and its equivalent network $\Sigma^{(k)}(0,m+1)$, we denote this value as $\alpha^{(k)}_{\mathrm{eq}(0)} = 1$
$\alpha^{(k)}_{x,i}$	Under the assumption that the buffer size is infinite, the optimal load fraction assigned to processor $p_{x,i}$ in $S^{i(k)}_{\mathrm{vac}}$ at iteration k. Similarly, for child processor $p_{i,j}$, we denote this value as $\alpha^{(k)}_{i,j}$
$L^{(k)}_{x,i}$	Amount of load assigned to $p_{x,i}$ at the kth iteration. Note that $L_{x,i} = \sum_{k=1}^{K} L^{(k)}_{x,i} = \sum_{k=1}^{K} \alpha^{(k)}_{x,i} Y^{(k)}$ is the total amount of load assigned to $p_{x,i}$
$T(\Sigma^{(k)}(x,i,m+1))$	This is defined as the optimal processing time of the assigned load fraction $\alpha^{(k)}_{x,\mathrm{eq}(i)}$ to $\Sigma^{(k)}(x,i,m+1)$ in the kth iteration. Note that for $\Sigma^{(k)}(0,m+1)$, we denote the optimal processing time as, $T(\Sigma^{(k)}(0,m+1)$
$T_{x,i}(\alpha)$	The time instant by which processor $p_{x,i}$ stops its computation under the distribution α
$T(\alpha)$	The total processing time of the entire load under the distribution α. Note that $T(\alpha) = \max\{T_{x,i}(\alpha)\}$ where the maximization is over all the processors in the network
$T^*(\alpha^*)$	The optimal processing time of a load, which is the minimum processing time to finish the processing of the entire load, using an optimal load distribution α^*
$S^{i(k)}_{\mathrm{vac}}$	The set of processors in $\Sigma^{(k)}(x,i,m+1)$ whose buffers are left unfilled at the start of the kth iteration. Thus, these are the processors available for processing at the kth iteration in $\Sigma^{(k)}(x,i,m+1)$
$S^{(k)}_{\mathrm{vac}}$	The set of processors in $G^{(k)}_T/G^{(k)}_{\mathrm{red}}$ whose buffers are left unfilled at the start of the kth iteration. Thus, these are the processors available for processing at the kth iteration in the entire tree. Note that $S^{(k)}_{\mathrm{vac}} = \cup_i S^{i(k)}_{\mathrm{vac}}$
$S^{(k)}_{\mathrm{filled}}$	The set of processors whose buffers are completely filled at the start of the kth iteration. Thus, the amount of load scheduled on these processors are equal to their initial buffer capacities, respectively
S_{buf}	The set of all processors in the network, i.e. $S_{\mathrm{buf}} = S^{(k)}_{\mathrm{vac}} + S^{(k)}_{\mathrm{filled}}, \forall k$
$X^{(k)}$	The amount of load left unscheduled at the start of the kth iteration. Initially, $X^{(1)} = L$
$Y^{(k)}$	The amount of load scheduled on the entire tree network at the kth iteration. Initially, $Y^{(0)} = 0$

property of IBS which states that the available buffer space at one or more processors may be completely filled during *every* iteration. Thus, the network size (in the sense of number of processors that participate) will be different as seen by the remaining unprocessed load. Consequently, this demands for computing the equivalent buffer capacity at a processor, especially if it is a parent processor. This is somewhat critical to our algorithm as every processor (especially an equivalent processor) has to 'declare' the amount of load that it can accommodate in this current iteration for processing. Thus, our approach in this paper recommends a 'pull–push' style of working as follows.

Every iteration involves two phases. In the first phase, referred to as *pull–phase*, it is imperative to determine the available resources. That is, the scheduler at the root processor must determine the total available buffer space in the network and the number of processors. Thus, in this phase, we first attempt to invoke procedures *Buffer Equivalence* (BUFF-EQ) and *Buffer Optimal*

Reduced Single-Level Tree Network (BUFF-O-R-SLTN) shown in Fig. 1, one after other, starting from the last level of the current G_T, say at iteration k ($G_\mathrm{T}^{(k)}$, as mentioned above). Fig. 1 presents the respective pseudo-codes for these procedures. These procedures are used to determine the equivalent buffer space that can be rendered by a sub-tree and its equivalent speed(s). Thus, every equivalent processor after this step will have two parameters associated with it namely, the equivalent speed and the equivalent buffer space available. It may be noted that in our current context, it may happen that either the parent processor or one of the child processors may not participate, and hence, the equivalent speed must be computed leaving these non-participating processors, for this single-level tree network under consideration. This process must percolate upwards till we reach the root processor, at which we obtain $p_{\mathrm{eq}(0)}^{(k)}$, which is the equivalent processor of $G_\mathrm{T}^{(k)}$ with an equivalent buffer of $B_{\mathrm{eq}(0)}^{(k)}$. Thus, at every iteration we first 'pull' the resulting $G_\mathrm{T}^{(k)}$ to reduce to an equivalent processor. In the second phase, referred to as *push-phase*, we invoke procedure *Load Distribution* (LD) shown in Fig. 2 to disseminate ('push') the intended load, $Y^{(k)} = B_{\mathrm{eq}(0)}^{(k)}$, starting from the root processor. Hence, we refer to this algorithm as *pull–push optimal load distribution* algorithm in this paper. Fig. 2 presents the corresponding pseudo-code for the procedure LD.

It must be stressed that algorithm PPOLD is an off-line algorithm. Thus, given an arbitrary tree network, the algorithm PPOLD first computes an optimal load distribution in off-line and then the scheduler residing

Procedure: **BUFF-EQ**(...)

Consider a single-level tree network $\Sigma^{(k)}(x, i, m+1)$ at a level, say q, with m child processors that belong to $S_{vac}^{i(k)}$, in the given tree $G_T^{(k)}$, at iteration k.

Step 1: Using following equations (1) and (2), we determine the optimal load fractions for this $\Sigma^{(k)}(x, i, m+1)$.

$$\alpha_{i,j}^{(k)} = \alpha_{i,m}^{(k)} \prod_{v=j+1}^{m} f_v, \; j = 0, 1, ..., m-1 \tag{1}$$

$$\text{where, } \alpha_{i,m}^{(k)} = \frac{1}{1 + \sum_{u=1}^{m} \prod_{v=u}^{m} f_v}, \tag{2}$$

$$f_v = \left(\frac{w_{i,v} + z_{i,v}\delta}{w_{i,(v-1)}} \right), \tag{3}$$

where $\delta = T_{cm}/T_{cp}$. Note that when $j = 0$, we refer to the parent processor $p_{x,i}^{(k)}$ with a load fraction $\alpha_{x,i}^{(k)} = \alpha_{i,0}^{(k)}$.

Step 2: An optimal amount of load that can be processed by $\Sigma^{(k)}(x, i, m+1)$ (or equivalently, an optimal buffer space that can be rendered) in this iteration k is given by, $Y_{x,i}^{(k)} = B_{x,eq(i)}^{(k)} = min\{B_{x,i}^{(k)}/\alpha_{x,i}^{(k)},\ B_{i,j}^{(k)}/\alpha_{i,j}^{(k)},\ \forall i, j \in S_{vac}^{i(k)}\}$.

Procedure: **BUFF-O-R-SLTN**(...)

Consider a single-level tree network $\Sigma^{(k)}(x, i, m+1)$ at a level, say q, with m child processors that belong to $S_{vac}^{i(k)}$, in the given tree $G_T^{(k)}$, at iteration k.

Step 1:

(i) As shown in Fig. 7(b), if the parent processor of this single-level tree is in $S_{vac}^{i(k)}$, then determine the equivalent processor $p_{x,eq(i)}^{(k)}$, with the equivalent speed $w_{x,eq(i)}^{(k)}$ given by A6 in the Appendix.

(ii) If the parent processor of this single-level tree is not in $S_{vac}^{i(k)}$, then we use A3 and A4 with $j = 0$ and $h = m$ to determine the equivalent speeds $w_{x,eq(i)}^{(k)}$ and $z_{x,eq(i)}^{(k)}$ of the child processors and links in $S_{vac}^{i(k)}$ of this single-level tree. Note that we assume that the root processor of this single-level tree can still communicate the load to its children with its front-ends, although it cannot participate in the computation. Also, note that in this case, the total communication delay from processor x to the children of i is simply given by, $(z_{x,eq(i)}^{(k)} + z_{x,i})$, where $z_{x,eq(i)}^{(k)}$ is given by A4, which is the equivalent link speed to be considered. This step is shown in Fig. 7(c).

Step 2: "Replace" the original single-level tree $\Sigma^{(k)}(x, i, m+1)$ with the above equivalent processor $p_{x,eq(i)}^{(k)}$ in the network $G_T^{(k)}$.

Fig. 1. Calculation of optimal equivalent buffer space and construction of optimal reduced single-level tree network.

Procedure: **LD**(...)

Starting from the equivalent tree $p^{(k)}_{eq(0)}$ with $\alpha^{(k)}_{eq(0)} = 1$, we do the following.

Step 1: Inflate $p^{(k)}_{eq(0)}$ to obtain a single-level tree $\Sigma^{(k)}(0, m+1)$ and distribute the load to all the $(m+1)$ processors in this tree. Note that there are m child processors in $\Sigma^{(k)}(0, m+1)$, in level 1.

Step 2: For every processor in a level $q, q = 1, 2, ..., Q$, we first verify whether it is an equivalent processor or a leaf processor (a processor with no children in the original G_T). For every equivalent processor i, in this level q, we simply inflate this equivalent processor and optimally distribute the load assigned to this equivalent processor i, by its parent, among the processors that formed this equivalent processor i. Note that this load distribution must be such that all the processors in this inflated equivalent processor i will stop computing at the same time instant, as per the optimality criterion, described by the following equations.

$$\alpha^{(k)}_{i,j} = \alpha^{(k)}_{i,m} \prod_{v=j+1}^{m} f_v, \; j = 0, 1, ..., m-1 \tag{4}$$

$$\text{where, } \alpha^{(k)}_{i,m} = \frac{\alpha^{(k)}_{x,eq(i)}}{1 + \sum_{u=1}^{m} \prod_{v=u}^{m} f_v}, \tag{5}$$

where f_v is as defined in (3) and $\alpha^{(k)}_{x,eq(i)}$ is the load assigned to $p^{(k)}_{x,eq(i)}$ by its parent processor. Also note that when $j = 0$, we refer to the parent processor $p^{(k)}_{x,i}$ with a load fraction $\alpha^{(k)}_{x,i} = \alpha^{(k)}_{i,0}$.

We terminate this step once all the equivalent processors in this level q are inflated.

Step 3: Let $q = q + 1$. If level $q \leq Q$, we repeat Step 2. Note that $q = Q$ corresponds to the last level in $G^{(k)}_T$.

Fig. 2. Load distribution procedure in $G^{(k)}_{\mathrm{T}}$.

at the root processor will carry out the actual load distribution process. A pseudo-code describing the working style of the entire algorithm PPOLD is presented in Fig. 3. Following numerical example illustrates the working of PPOLD in seeking an optimal load distribution.

Example 1. (PPOLD) Consider a tree network G_T with 2 levels, 8 processors and 7 links with the following speed parameters. As shown in Fig. 4(a), the processor speeds are $w_0 = 1$, $w_{0,1} = 1$, $w_{0,2} = 2$, $w_{0,3} = 1$, $w_{0,4} = 1$, $w_{0,5} = 1$, $w_{2,1} = 2$, $w_{2,2} = 3$; link speeds are $z_{x,i} = z = 1$; and the available buffer capacity at each $p_{x,i} \in G_T$ is $B_0 = 35.000$, $B_{0,1} = 15.000$, $B_{0,2} = 10.000$, $B_{0,3} = 25.000$, $B_{0,4} = 10.000$, $B_{0,5} = 15.000$, $B_{2,1} = 5.000$, $B_{2,2} = 5.000$, respectively. Further, we let $L = 100$. Note that Rule A is satisfied by all the processor-link pairs in the tree. Thus, we can directly apply PPOLD to get the load distribution results which are listed in Table 2.

As shown in Table 2, we obtain an optimal load distribution (by employing all the above three procedures recursively as explained on each single-level tree in the network shown in Fig. 4(b) through which every processor in the tree was given its optimal load fraction: $L^*_0 = 35.000$, $L^*_{0,1} = 15.000$, $L^*_{0,2} = 10.000$, $L^*_{2,1} = 5.000$, $L^*_{2,2} = 5.000$, $L^*_{0,3} = 17.143$, $L^*_{0,4} = 8.571$, $L^*_{0,5} = 4.286$. Thus, starting from the root processor p_0, the scheduler at every parent processor shall distribute these load fractions to their respective child processors, to obtain the minimum processing time. Note that the entire distribution procedure should be carried out in a way that is consistent with the network structure. This means that, for instance, at first, p_0 will distribute $L^*_{0,1}$ to $p_{0,1}$, $(L^*_{0,2} + L^*_{2,1} + L^*_{2,2})$ as a whole to $p_{0,2}$ next, $L^*_{0,3}$, $L^*_{0,4}$, and $L^*_{0,5}$ to processors $p_{0,3}$, $p_{0,4}$, $p_{0,5}$, respectively, and compute its own portion L^*_0 concurrently. Next, scheduler at $p_{0,2}$ will do the same job as p_0 after the above load $(L^*_{0,2} + L^*_{2,1} + L^*_{2,2})$ arrives. It then transfers $L^*_{2,1}$, and $L^*_{2,2}$ to its child processors $p_{2,1}$ and $p_{2,2}$, respectively. Further, it should be noted that although it may take several iterations to compute an optimal load fraction for a processor (since a processor may participate in many iterations), during the load distribution phase, the cumulative load to be processed by a processor is assigned at a time. Thus, for instance, in our example, processor $p_{0,3}$ participates in all the iterations, whereas all its cumulative load (17.143) is transferred to it at one time, as shown in the timing diagram (Fig. 4(c)). The optimal processing time for the entire load is then

```
Procedure: **PPOLD**(...)
**Initial state:**
  $S^{(0)}_{vac} = S_{buf}$; $S^{(0)}_{filled} = \Phi$; $X^{(0)} = L$; $Y^{(0)} = 0$; initial buffer capacities are
  $B^{(0)}_{x,i}$, $\forall p_{x,i} \in G_T$;
  if ($\sum_{\forall p_{x,i}} B^{(0)}_{x,i} < L$), exit with error message "Error! Not enough buffer
  capacity to process any load greater than $\sum_{\forall p_{x,i}} B^{(0)}_{x,i}$";
  if ($\sum_{\forall p_{x,i}} B^{(0)}_{x,i} = L$), exit with optimal solution $\alpha^*$, by which $L^*_{x,i} = B^{(0)}_{x,i}$;
  Set $k = 1$;
**Do {**
  **Phase I:** Resetting the variables
    $X^{(k)} = X^{(k-1)}$; $S^{(k)}_{vac} = S^{(k-1)}_{vac}$; $S^{(k)}_{filled} = S^{(k-1)}_{filled}$; $B^{(k)}_{x,i} = B^{(k-1)}_{x,i}$;
  **Phase II:** Scheduling
    *Step 1. Find out the portion of load $Y^{(k)}$ which should be scheduled in*
    *$G^{(k)}_T$ at the k-th iteration:*
      for ($q = Q; q > 0; q - -$) /* $Q$: total number of levels in $G_T$
        {
          for ($r = 1; r \le R; r + +$) /* $R$: total number of sub-trees at level $q$
            {
              $B^{(k)}_{x,eq(i)}$ = BUFF-EQ(...); /* Refer to Fig. 1
              $p^{(k)}_{x,eq(i)}$ = BUFF-O-R-SLTN(...); /* Refer to Fig. 1
            }
        }
      if ($B^{(k)}_{eq(0)} > X^{(k)}$), $Y^{(k)} = X^{(k)}$;
      else $Y^{(k)} = B^{(k)}_{eq(0)}$;
    *Step 2. Schedule $Y^{(k)}$ among all the processors $p_{x,i} \in S^{(k)}_{vac}$:*
      $\alpha^{(k)}$ = LD(...), by which $L^{(k)}_{x,i} = \alpha^{(k)}_{x,i} \cdot Y^{(k)}$; /* Refer to Fig. 2
  **Phase III:** Updating the variables
    $X^{(k)} = X^{(k)} - Y^{(k)}$;
    if ($X^{(k)} = 0$), break;
    else
      {
        for ($\forall p_{x,i} \in S^{(k)}_{vac}$), { if $B^{(k)}_{x,i} = \alpha^{(k)}_{x,i} \cdot Y^{(k)}$, $S^{(k)}_{filled} = S^{(k)}_{filled} \cup \{p_{x,i}\}$ };
        $S^{(k)}_{vac} = S_{buf} - S^{(k)}_{filled}$;
        for ($\forall p_{x,i} \in S^{(k)}_{vac}$), $B^{(k)}_{x,i} = B^{(k)}_{x,i} - \alpha^{(k)}_{x,i} \cdot Y^{(k)}$;
      }
    Set $k = k + 1$;
**} while** ($k \le N$)
**Final Solution:** $L_{x,i} = \sum_{k=1}^{K} \alpha^{(k)}_{x,i} \cdot Y^{(k)}$; /* $K \le N$: total number of iterations
**Processing Time:** $T(\alpha) = maxT_{x,i}(\alpha)$.
```

Fig. 3. Algorithm PPOLD.

given by, 69.286. Timing diagram shown in Fig. 4(c) demonstrates the exact load distribution process and shows *how* and *when* these load fractions are to be communicated within the network and computed at each processor.

Furthermore, it is worth mentioning that PPOLD is a natural extension of IBS for generic tree networks whereas IBS is restricted to single-level tree networks. Also, PPOLD preserves all the features of IBS. All the properties (lemmas) proved for IBS in Ref. [18] also hold for PPOLD. Thus, it should come in no surprise to realize that PPOLD can yield the same optimal solution generated by IBS for a single-level tree network, as after all a single-level tree network is a special case of general tree networks.

3.2. Strategies for load distribution in G_T: case (b)

In this section, we consider extending the problem for a generic case in which Rule A need not be satisfied by all the processor-link pairs in the network. We propose three heuristic strategies that are based on certain criteria. The first strategy (Strategy I) basically attempts to use Rule A and identifies the redundant processor-link pairs. It carefully attempts to make use of the eliminated redundant processor-link pairs in the previous iterations in the current iteration. This is done by a systematic check on Rule A in the entire network. It may be noted that our earlier assumption (that the total buffer capacity of the processors which satisfy Rule A in G_T is strictly greater than L) fully supports the idea employed in this strategy.

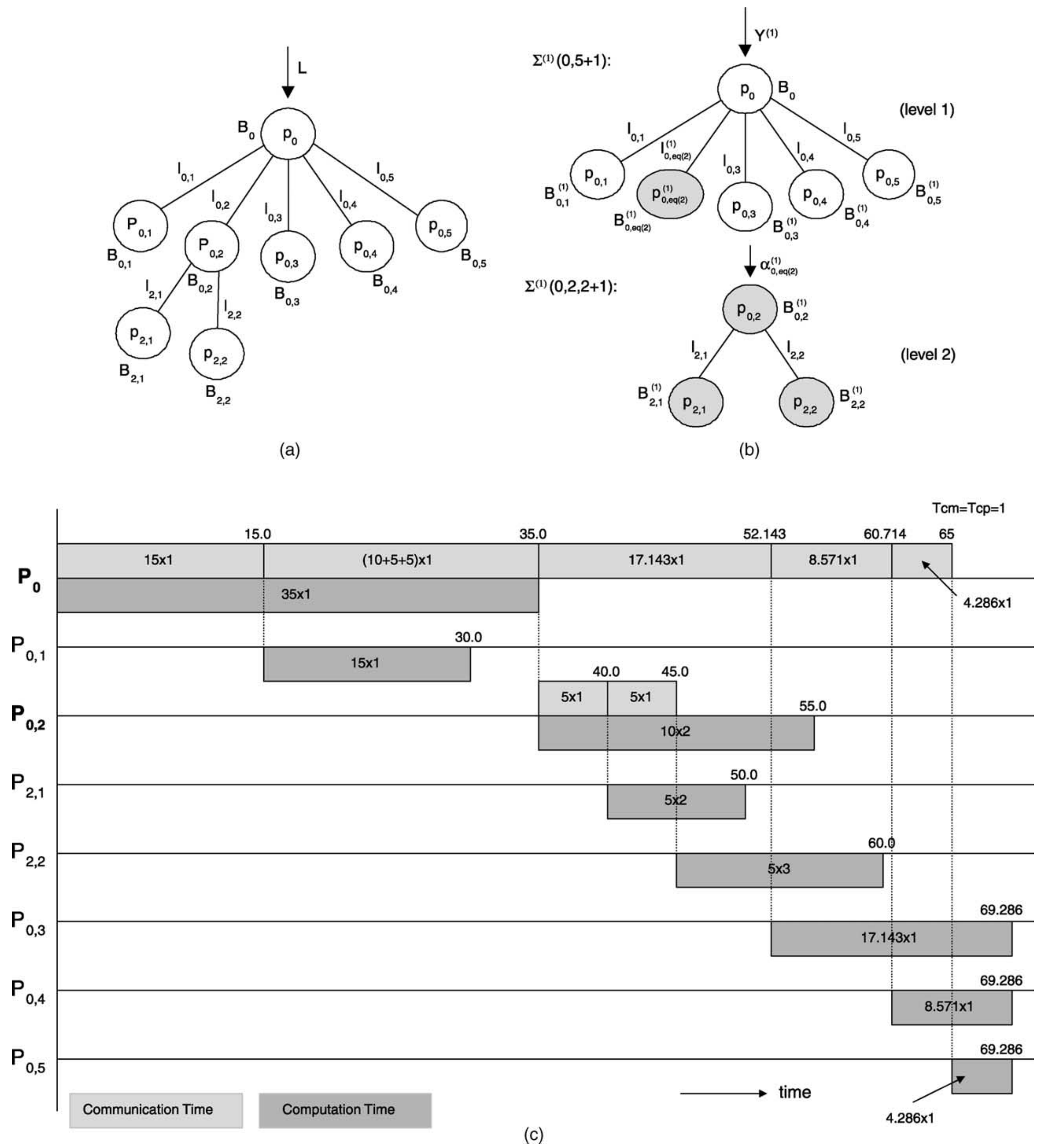

Fig. 4. Example 1. (a) A two-level G_T tree, (b) sub-trees as seen by parent processors; (c) timing diagram showing an optimal load distribution.

The second and third strategies we propose make use of the optimal sequencing theorem.

Strategy I: Strategy I is designed to dynamically obtain a reduced network, referred to as $G^{(k)}_{\text{red}}$, by examining the redundancy of $G^{(k)}_{\text{T}}$ in every iteration. That is, in every iteration, we shall apply Rule A during the construction of $G^{(k)}_{\text{red}}$ to obtain the 'best' set of processors to distribute for this particular iteration. We denote the set of redundant processor-link pairs eliminated so far as RS. Further, before applying Rule A in every single-level tree, we consider including the set of processor-link pairs from RS in this current iteration for possible participation in the computation. After including these redundant processor-link pairs, we apply Rule A to determine the actual set of processor-link pairs that will participate in this current iteration. As we can observe, the participating redundant processor-link pairs from RS may not violate Rule A in the current iteration.

Table 2
Simulation results of Example 1

k	$X^{(k)}$	$B_0^{(k)}$	$B_{0,1}^{(k)}$	$B_{0,2}^{(k)}$	$B_{2,1}^{(k)}$	$B_{2,2}^{(k)}$	$B_{0,3}^{(k)}$	$B_{0,4}^{(k)}$	$B_{0,5}^{(k)}$	$Y^{(k)}$
Initial	100.000	35.000	15.000	10.000	5.000	5.000	25.000	10.000	15.000	0.000
1	100.000	35.000	15.000	10.000	5.000	5.000	25.000	10.000	15.000	59.062
2	40.938	5.000	0.000	6.250	2.500	3.750	21.250	8.125	14.063	9.688
3	31.250	0.000	0.000	5.000	1.667	3.333	20.000	7.500	13.750	9.375
4	21.875	0.000	0.000	2.500	0.000	2.500	17.500	6.250	13.125	8.125
5	13.750	0.000	0.000	0.000	0.000	1.250	15.000	5.000	12.500	4.531
6	9.219	0.000	0.000	0.000	0.000	0.000	13.125	4.063	12.031	9.219
Final	0.000	0.000	0.000	0.000	0.000	0.000	7.857	1.429	10.714	–

	p_0	$p_{0,1}$	$p_{0,2}$	$p_{2,1}$	$p_{2,2}$	$p_{0,3}$	$p_{0,4}$	$p_{0,5}$	
$L_{x,i}^*$	35.000	15.000	10.000	5.000	5.000	17.143	8.571	4.286	$L = 100.000$
$T_{x,i}^*(\alpha^*)$	35.000	30.000	55.000	50.000	60.000	69.286	69.286	69.286	$T^*(\alpha^*) = 69.286$

This will particularly happen if those processors which participated in earlier iterations cannot participate in the current iteration any longer due to non-availability of buffer space. Consequently, these processors, whose buffers are filled, should not be considered as active participants and hence, processors from set RS may now qualify in the computation process. Similarly, it may happen that some of the processor-link pairs that are so far active (until last iteration) may not participate in the current iteration due to similar reasons mentioned above. To implement this idea, we shall follow the same procedure as BUFF-EQ described in PPOLD except that we take into account of Rule A in every iteration and we refer to it as *Buffer Equivalence with Rule A* (BUFF-EQ-RULE$_A$) shown in Fig. 5.

The usefulness of Strategy I is significant when the size of RS set and the depth (number of levels) of the network are large, as it attempts to utilize the existing resources in the best possible way at the right iterations. Certainly, the overheads in carrying out Strategy I has to be carefully weighed against its use in any network, if not the cost of running Strategy I may become a bottleneck.

Strategy II: From resource utilization perspective, with the repeated application of Rule A, Strategy I attempts to deliver the best performance. However, in the literature, for single-level tree networks, the concept of sequencing has been proved to deliver an optimal performance [5]. Thus, with PPOLD as a basis, we attempt to consider the possibility of applying the optimal sequencing theorem (as stated below) for load distribution in G_T. This is done by applying the optimal sequencing theorem for every sub-tree network in G_T. For the purpose of continuity, here, we state the optimal sequencing theorem quoted in Ref. [5] for a single-level tree network.

Theorem 1. (*Optimal Sequence*) *In a single-level tree* $\Sigma^{(k)}(x, i, m+1)$, *in order to achieve minimum processing time, the sequence of load distribution by the root processor* $p_{x,i}$ *should follow the order in which the link speeds* $1/z_{i,j}$, $j = 1, 2, \ldots, m$, *decrease.*

Thus, in our context, the above theorem is applied as follows. In the original network G_T, we systematically apply Theorem 1 to identify the optimal sequence in every single-level tree network, starting from the last level to the root processor. This is important to identify

Procedure: **BUFF-EQ-RULE**$_A$(...)

Consider a single-level tree network $\Sigma^{(k)}(x, i, m+1)$ at a level, say q, with m child processors that belong to $S_{vac}^{i(k)}$, in the given tree $G_T^{(k)}$, at iteration k.

Step 1: Apply Rule A as described in the Appendix and eliminate any redundant processor-link pairs that violate condition A1. By doing so, we can get an optimal reduced single-level tree network $\Sigma^{(k)}(x, i, n+\ )$ where $n \le m$ [5] for distribution purpose in this particular iteration k.

Step 2: Using (1) and (2), we determine the optimal load fractions for this $\Sigma^{(k)}(x, i, n+1)$.

Step 3: An optimal amount of load that can be processed by $\Sigma^{(k)}(x, i, n+1)$ (or equivalently, an optimal buffer space that can be rendered) in this iteration k is given by, $Y_{x,i}^{(k)} = B_{x,eq(i)}^{(k)} = min\{B_{x,i}^{(k)}/\alpha_{x,i}^{(k)},\ B_{i,j}^{(k)}/\alpha_{i,j}^{(k)},\ \forall i, j \in S_{vac}^{i(k)}\}$.

Fig. 5. Calculation of equivalent buffer space for iteration k in a single-level tree network for Strategy I.

Procedure: **BUFF-EQ-OS**(...)

Consider a single-level tree network $\Sigma^{(k)}(x, i, m+1)$ at a level, say q, with m child processors that belong to $S^{i(k)}_{vac}$, in the given tree $G^{(k)}_T$, at iteration k.

Step 1: For Strategy II: If $k = 1$, identify the optimal sequence of load distribution by applying Theorem 1. For all $k \geq 2$, go to Step 2 directly.

For Strategy III: Identify an optimal sequence of load distribution by applying Theorem 1.

Step 2: Using (1) and (2), we determine the optimal load fractions for this $\Sigma^{(k)}(x, i, m+1)$.

Step 3: An optimal amount of load that can be processed by $\Sigma^{(k)}(x, i, m+1)$ (or equivalently, an optimal buffer space that can be rendered) in this iteration k is given by, $Y^{(k)}_{x,i} = B^{(k)}_{x,eq(i)} = min\{B^{(k)}_{x,i}/\alpha^{(k)}_{x,i},\ B^{(k)}_{i,j}/\alpha^{(k)}_{i,j},\ \forall i, j \in S^{i(k)}_{vac}\}$.

Fig. 6. Calculation of equivalent buffer space for iteration k in a single-level tree network for strategies II and III.

as every parent processor will follow this optimal sequence while distributing the load. Thus, every parent processor will apply Theorem 1 and obtain the optimal sequence. However, as we know, the above optimal sequence holds in infinite-buffer case cannot be directly used in finite-buffer context. Thus, it is natural to expect that the optimal sequencing theorem may not fetch an optimal solution in our problem context; however, one may expect a performance gain. We revisit procedure BUFF-EQ to include the optimal sequencing property and we refer to this procedure as *Buffer Equivalence with Optimal Sequencing* (BUFF-EQ-OS) (Fig. 6) and use it in Strategy II. Finally, it may be noted that, we apply the optimal sequencing theorem for every sub-tree and determine the optimal load distribution.

Strategy III: Although Strategy II uses optimal sequencing concept, the way in which the load is distributed is off-line. That is, Strategy II will run in off-line and the resulting loads for individual processors are distributed in a single-installment. Further, in Strategy II, we use the optimal sequencing theorem only once and compute the load distribution. However, we will now show that for applications that demand on-line processing of loads, Strategy II can be tuned to work dynamically. This is done in such a way that for every iteration the optimal sequencing theorem can be applied for every sub-tree and after every iteration, the respective load fractions can be disseminated to the processors, then the processing time can be further improved. Further, in this strategy, it may be realized that, in every iteration, the optimal sequence in a sub-tree in G_T may change. This may happen when the equivalent link speeds get changed during the PULL phase in PPOLD. For instance, in Step 1(ii) of BUFF-O-R-SLTN when the parent processor $p_{x,i}$ in $\Sigma^{(k)}(x, i, m+1)$ gets completely filled in iteration k, the equivalent link speed $z^{(k)}_{x,\text{eq}(i)}$ becomes $(z^{(k)}_{x,\text{eq}(i)} + z_{x,i})$. Hence, for iteration k, Theorem 1 will recommend an *optimal sequence* which is different from the one of previous iterations, for a sub-tree rooted at $p_{x,i}$. Thus, we clearly see that in every iteration we need to identify an optimal sequence and then determine the respective load distribution. We refer to BUFF-EQ-OS shown in Fig. 6 for Strategy III to compute the equivalent buffer space.

Remarks: It may be possible that there could exist a slight dilemma on why we are concerned with the design of an on-line version of Strategy III alone and not with other strategies. The answer to this dilemma can be resolved if we observe that the on-line working style of this strategy evolves naturally. This is due to the fact that in every iteration we obtain an optimal sequence and hence, it is mandatory that the load distribution must also follow the same sequence in every iteration. Consequently, the algorithm naturally works in on-line. Thus, Strategy III is a natural choice to handle real-time (deadline driven) processing demands. In the following example, we will demonstrate the workings of all the three strategies proposed above.

Example 2. Consider a tree network G_T with 2 levels, 8 processors and 7 links, as shown in Fig. 8. Following are the speed parameters assumed. The processor speeds are $w_0 = 1$, $w_{0,1} = 1$, $w_{0,2} = 2$, $w_{0,3} = 1$, $w_{0,4} = 3$, $w_{2,1} = 1$, $w_{2,2} = 1$, $w_{2,3} = 2$; link speeds are $z_{0,1} = 0.6$, $z_{0,2} = 0.3$, $z_{0,3} = 0.1$, $z_{0,4} = 0.8$, $z_{2,1} = 0.8$, $z_{2,2} = 0.1$, $z_{2,3} = 0.6$ and the available buffer capacity at each processor $p_{x,i} \in G_T$ is $B_0 = 35.000$, $B_{0,1} = 25.000$, $B_{0,2} = 10.000$, $B_{0,3} = 5.000$, $B_{0,4} = 20.000$, $B_{2,1} = 10.000$, $B_{2,2} = 5.000$ and $B_{2,3} = 30.000$, respectively. Further, we let $L = 100$. We assume that the fixed sequence of load distribution used in Strategy I is from the left hand side to the right hand side in every single-level tree in this case.

Firstly, as shown in Table 3, the processing time of Strategy I is 46.747. As mentioned in Strategy I, it may be possible that we can apply Rule A only once and distribute the load to the resulting 'fixed' G_{red}, however, in this case, we note that the processing time is 88.250. Thus, comparing these two cases, we observe that the processing time decreases by almost 50% by this

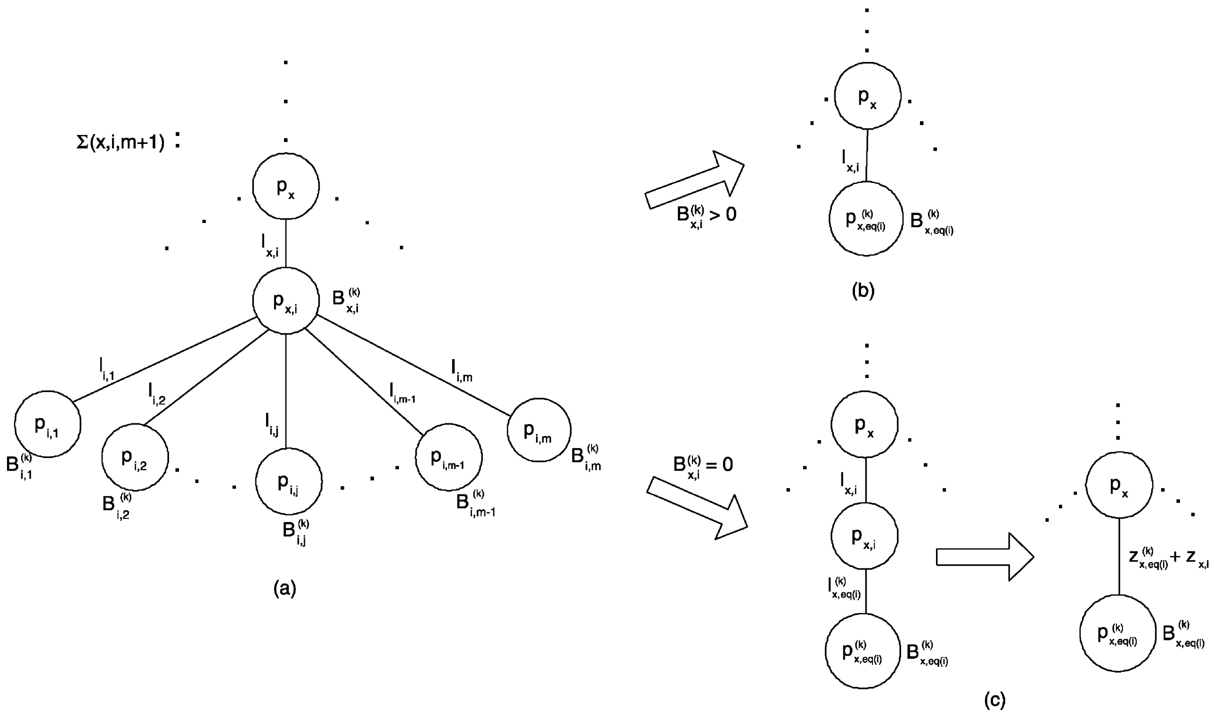

Fig. 7. Determining an equivalent processor in finite-buffer case (a) a single-level tree $\Sigma(x, i, m+1)$ in G_T; (b) Step1(i) of BUFF-O-R-SLTN; (c) Step1(ii) of BUFF-O-R-SLTN.

dynamic application of Rule A in Strategy I. It may be noted that $p_{0,1}$ and $p_{2,1}$ violate Rule A in the original tree G_T. Using Rule A in the first iteration, these two processors will be excluded for this iteration, however, from iteration 2 onwards, $p_{0,1}$ and $p_{2,1}$ no longer violate Rule A. This is because the buffers of $p_{0,3}$ and $p_{2,2}$ are completely utilized in iteration 1, and hence, they will not be participating after iteration 1 and are thus no longer available in the subsequent iterations. Consequently, we include processors $p_{0,1}$ and $p_{2,1}$ in the subsequent iterations. Readers could refer to the timing diagram shown in Fig. 9.

Secondly, the processing time given by Strategy II is 44.602 and Table 4 presents the load distribution obtained. Table 5 presents the load distribution for Strategy III and the processing time is given by 35.000. Comparing these two strategies, we clearly observe that Strategy III outperforms Strategy II, as expected. Readers could refer to timing diagrams shown in Figs. 10 and 11 for a detailed understanding. However, at this

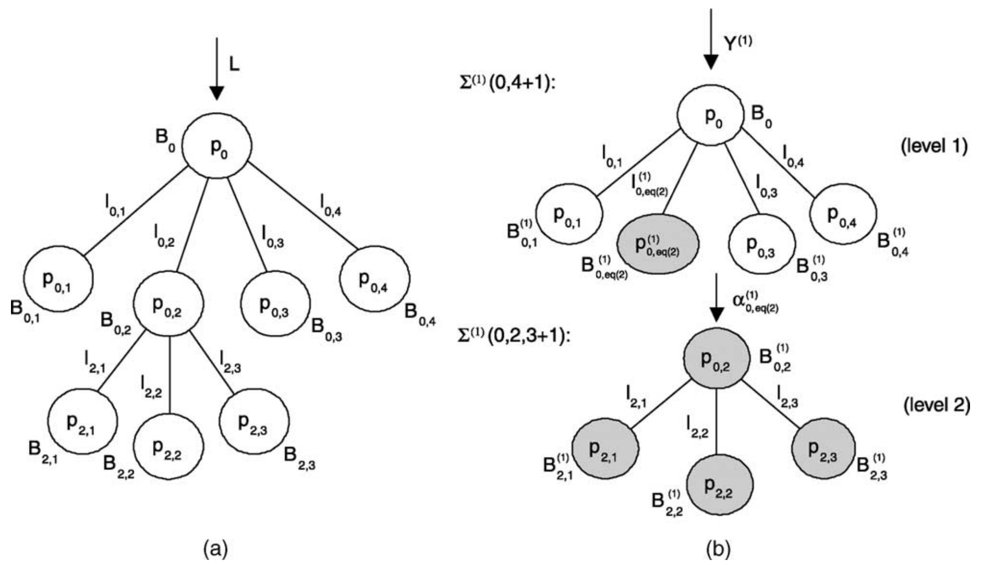

Fig. 8. Example 2. (a) A two-level G_T tree; (b) sub-trees as seen by parent processors.

Table 3
Simulation results of Strategy I (Example 2)

k	$X^{(k)}$	$B_0^{(k)}$	$B_{0,1}^{(k)}$	$B_{0,2}^{(k)}$	$B_{2,1}^{(k)}$	$B_{2,2}^{(k)}$	$B_{2,3}^{(k)}$	$B_{0,3}^{(k)}$	$B_{0,4}^{(k)}$	$Y^{(k)}$
Initial	100.000	35.000	25.000	10.000	10.000	5.000	30.000	5.000	20.000	0.000
1	100.000	35.000	25.000	10.000	10.000	5.000	30.000	5.000	20.000	24.391
2	75.609	26.598	25.000	7.250	10.000	0.000	28.077	0.000	18.684	61.671
3	13.938	0.000	8.376	1.230	3.311	0.000	25.504	0.000	15.516	7.168
4	6.770	0.000	4.979	0.000	1.9444	0.000	24.979	0.000	14.868	6.770
Final	0.000	0.000	1.034	0.000	0.164	0.000	24.294	0.000	14.508	–
	p_0	$p_{0,1}$	$p_{0,2}$	$p_{2,1}$	$p_{2,2}$	$p_{2,3}$	$p_{0,3}$	$p_{0,4}$		
$L_{x,i}$	35.000	23.966	10.000	9.836	5.000	5.706	5.000	5.492	$L = 100.000$	
$T_{x,i}(\alpha)$	35.000	38.345	43.542	41.247	36.911	46.747	29.042	44.912	$T(\alpha) = 46.747$	

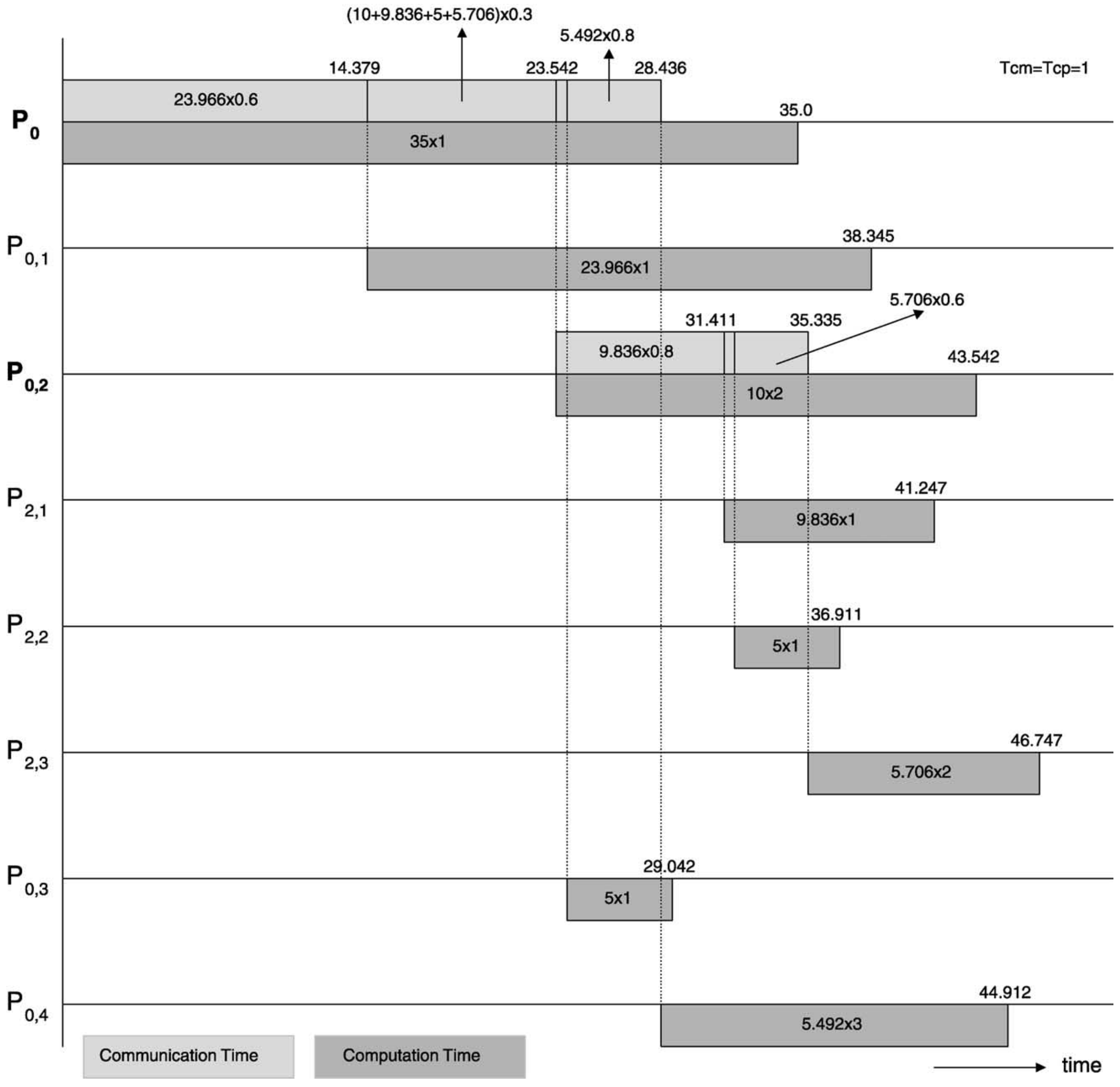

Fig. 9. Timing diagram of Strategy I (Example 2).

Table 4
Simulation results of Strategy II (Example 2)

k	$X^{(k)}$	$B_0^{(k)}$	$B_{0,1}^{(k)}$	$B_{0,2}^{(k)}$	$B_{2,1}^{(k)}$	$B_{2,2}^{(k)}$	$B_{2,3}^{(k)}$	$B_{0,3}^{(k)}$	$B_{0,4}^{(k)}$	$Y^{(k)}$
Initial	100.000	35.000	25.000	10.000	10.000	5.000	30.000	5.000	20.000	0.000
1	100.000	35.000	25.000	10.000	10.000	5.000	30.000	5.000	20.000	19.431
2	80.569	29.500	23.099	8.479	8.819	2.235	28.937	0.000	19.500	11.264
3	69.305	25.457	21.563	7.25	7.863	0.000	28.077	0.000	19.095	50.678
4	18.627	5.250	12.500	0.000	1.667	0.000	22.500	0.000	16.711	9.332
5	9.295	0.400	11.458	0.000	0.000	0.000	21.000	0.000	16.436	0.756
6	8.539	0.000	11.286	0.000	0.000	0.000	20.862	0.000	16.391	8.539
Final	0.000	0.000	7.147	0.000	0.000	0.000	17.551	0.000	15.302	–
	p_0	$p_{0,1}$	$p_{0,2}$	$p_{2,1}$	$p_{2,2}$	$p_{2,3}$	$p_{0,3}$	$p_{0,4}$		
$L_{x,i}$	35.000	17.853	10.000	10.000	5.000	12.449	5.000	4.698	$L = 100.000$	
$T_{x,i}(\alpha)$	35.000	40.299	31.735	37.704	17.235	44.602	5.500	40.299	$T(\alpha) = 44.602$	

stage, we can clearly sense that there exists a trade-off on the choice of using strategies. We will discuss this in Section 4.

4. Discussions of the results

Although scheduling divisible loads on tree networks has been addressed in the DLT literature, in this paper, a complete and rigorous treatment is carried out in a systematic fashion. We have explicitly presented all the procedures that would be used in a real-life situations while scheduling a divisible load on tree networks. In addition, we consider the constraints imposed by the processors in accommodating only a finite amount of divisible load for processing, as it has finite buffer space to render. As mentioned in Section 1, study in Ref. [18] is the first attempt to consider the scheduling problem under buffer capacity constraints. In this paper, all the proposed algorithms attempt to make use of IBS algorithm in a systematic fashion for arbitrary tree networks. Further, it has been pointed out in Refs. [5,10] that in a tree network, not all processors need to participate in the computation, as inclusion of redundant processors may penalize the performance. Thus, Rule A, for single-level tree networks in Ref. [5], elegantly proposes a methodology by which such redundant processor-link pairs can be eliminated from computation. Rule A, in a strict sense, is proved to guarantee an optimal solution for single-level tree networks, and our approach in this paper considers single-level trees in every step. Further, our algorithms are designed to work in a *bottom-up* fashion and hence, every step guarantees an optimal performance, if it can be achieved.

Coupled with the applicability of Rule A, we have used IBS to render an optimal amount of buffer space required for computation in every iteration. Thus, every step, in our algorithms, involves a systematic way of determining the amount of buffer space required (also equivalent to saying that the amount of load to be scheduled in that iteration) and the number of processors participating in the network. The examples presented in this paper demonstrate the workings of the algorithms and elicit certain important properties. Algorithm PPOLD proposed in this paper works as a common methodology for both the cases (a) and (b), as it is mainly concerned

Table 5
Simulation results of Strategy III (Example 2)

k	$X^{(k)}$	$B_0^{(k)}$	$B_{0,1}^{(k)}$	$B_{0,2}^{(k)}$	$B_{2,1}^{(k)}$	$B_{2,2}^{(k)}$	$B_{2,3}^{(k)}$	$B_{0,3}^{(k)}$	$B_{0,4}^{(k)}$	$Y^{(k)}$
Initial	100.000	35.000	25.000	10.000	10.000	5.000	30.000	5.000	20.000	0.000
1	100.00	35.000	25.000	10.000	10.000	5.000	30.000	5.000	20.000	19.431
2	80.569	29.500	23.099	8.479	8.819	2.235	28.937	0.000	19.500	11.264
3	69.305	25.457	21.563	7.250	7.863	0.000	28.077	0.000	19.095	50.678
4	18.627	5.250	12.500	0.000	1.667	0.000	22.500	0.000	16.711	11.086
5	7.541	0.000	9.219	0.000	0.776	0.000	21.699	0.000	15.847	5.091
6	2.450	0.000	6.357	0.000	0.000	0.000	21.000	0.000	15.094	2.450
Final	0.000	0.000	4.761	0.000	0.000	0.000	20.565	0.000	14.674	–
	p_0	$p_{0,1}$	$p_{0,2}$	$p_{2,1}$	$p_{2,2}$	$p_{2,3}$	$p_{0,3}$	$p_{0,4}$		
$L_{x,i}$	35.000	20.239	10.000	10.000	5.000	9.435	5.000	5.326	$L = 100.000$	
$T_{x,i}(\alpha)$	35.000	34.776	27.036	28.693	8.043	30.904	5.500	33.145	$T(\alpha) = 35.000$	

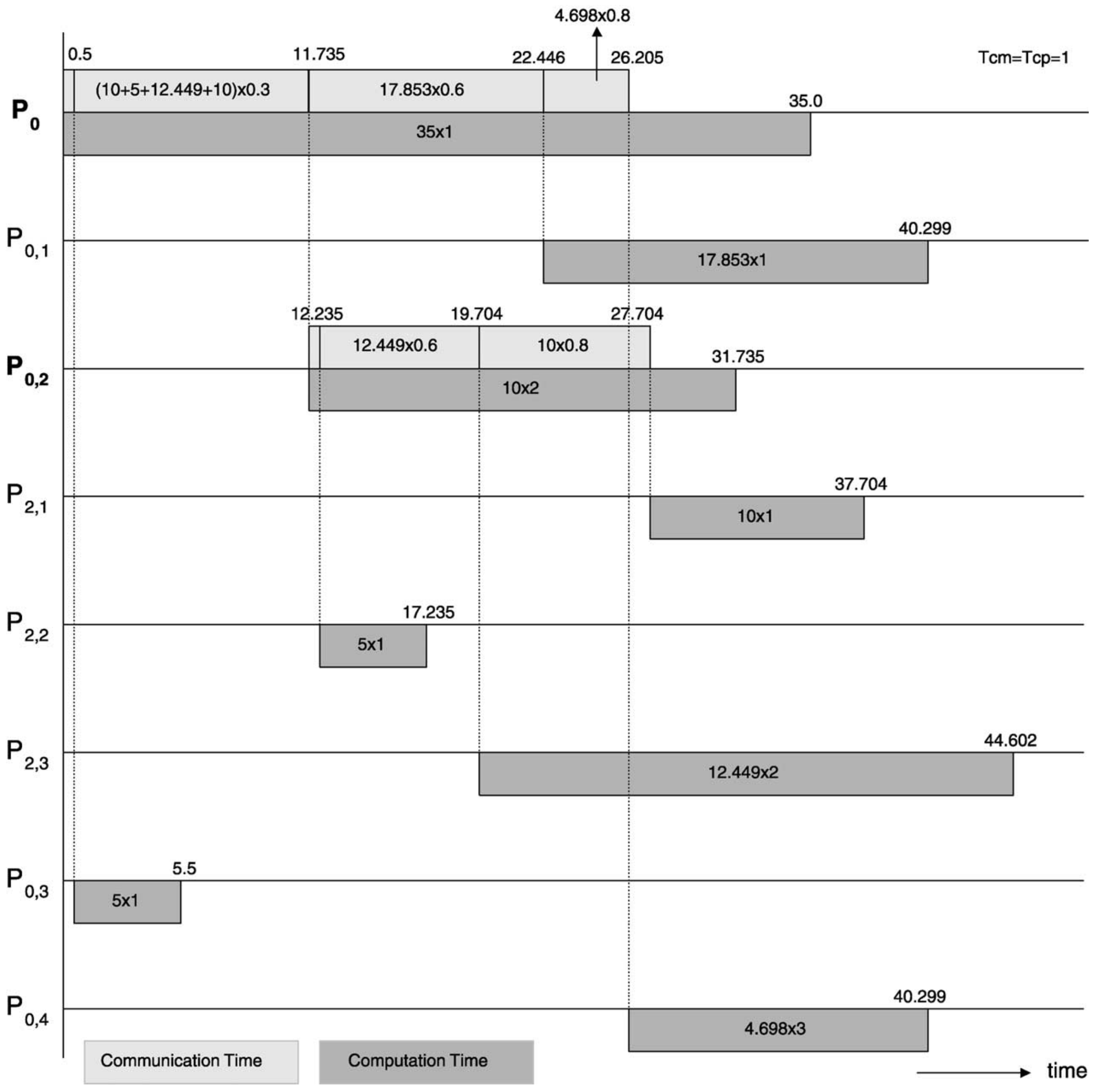

Fig. 10. Timing diagram of Strategy II (Example 2).

with load distribution. Thus, in a network, when all the processor-link pairs satisfy Rule A, then for a given fixed sequence, PPOLD yields an optimal solution. This is shown in Example 1. Complete workings of strategies I, II and III are demonstrated in Example 2. It may be noted that although we use Rule A and IBS in Strategy I, the fact that these concepts are employed independently may prevent Strategy I from delivering an optimal performance. In fact, even with strategies II and III, the concept of optimal sequencing and IBS are employed independently and hence, this may prevent yielding optimal solutions with these strategies. We shall exactly compute the complexities of these strategies in Section 4.1 and then highlight the applicability of the strategies under various situations.

4.1. Calculation of complexities of the strategies

We shall now quantify the complexities of all the algorithms proposed in this paper. In our calculation, we first compute the number of steps involved in one iteration, where one iteration refers to the computation involved for the entire tree network—typically, a pull and a push phase. Assuming that there are m processors in every sub-tree and that there are R sub-trees in every level, with a total number of Q levels in the entire tree network, the worst case complexity of the pull phase of PPOLD, for an iteration, can be readily seen as $O((m+1)RQ) = O(N + RQ)$ steps, where N is the total number of processors in the network. The factor m comes because of the fact that we must account for determining equivalent buffer required

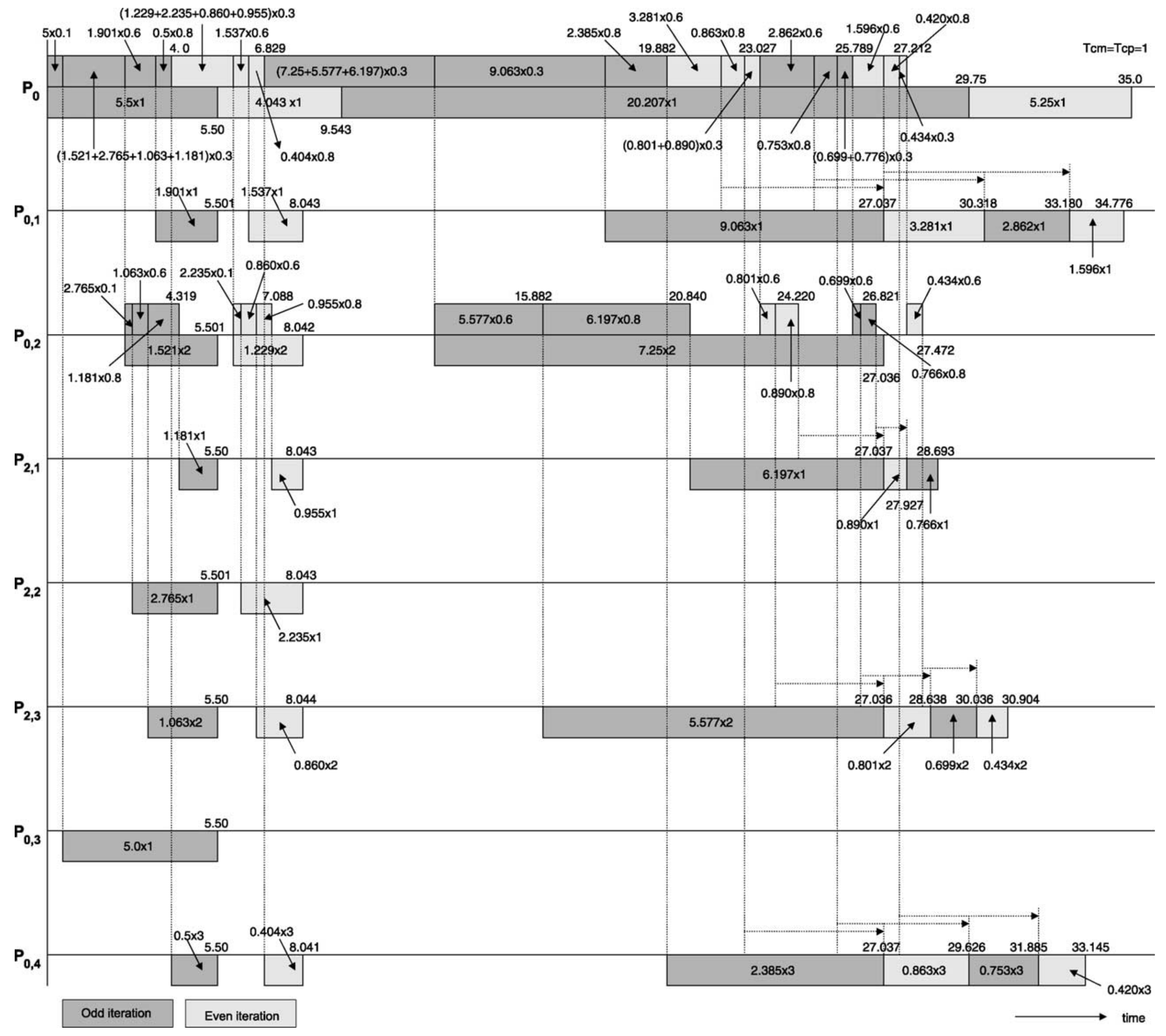
Fig. 11. Timing diagram of Strategy III (Example 2).

(and the computation of equivalent processor-link is done in O(1) using Eq. (A6) for each sub-tree). For the push phase, the complexity would be $O(N)$ steps, in an iteration. Thus, for every iteration, the total complexity is $O(2N + RQ)$. However, since IBS guarantees that at least one buffer will be completely filled in every iteration, the entire algorithm PPOLD will run for at most $O(N)$ iterations, and hence, the overall complexity of PPOLD algorithm is given by $O(N(2N + RQ) = O(2N^2 + NRQ)$. The best case complexity would be when each buffer has a capacity greater than or equal to L, and hence, procedure BUFF-EQ can be avoided altogether. Thus, in this case, the overall complexity of PPOLD will be at most $O(2N + RQ)$ (with only one iteration needed). Although case (a) assumes that Rule A is satisfied by all the processor-link pairs in the network, in real-life situation, it may take at most another $O(N)$ steps to validate this assertion. Note that this is done only once at the start of PPOLD. However, it should bring no surprise that the above mentioned complexities for the worst and best cases would not be affected by this validation procedure.

Strategy I of case (b) carries out the above mentioned validation of Rule A in every iteration. This is because, in Strategy I, we attempt to render additional buffer available from redundant processors that were eliminated in the previous iterations. Thus, it becomes imperative to validate Rule A in every iteration. Hence, the complexity of Strategy I would be $O(3N^2 + NRQ)$, by following the same line of argument as above. The key difference lies in validating Rule A in every iteration that consumes additional $O(m)$ steps per iteration per sub-tree. Similarly, Strategy II has a complexity of $O(2N^2 + NRQ + RQ \log m)$, and for Strategy III, this would be $O(2N^2 + NRQ + NRQ \log m)$, respectively. The difference in the complexities of strategies

PPOLD and II (and III) is because of sequencing (consuming O(log m) steps for each sub-tree with m processors per iteration). An additional factor N, in the last component of Strategy III's complexity calculation, is due to the fact that we apply the optimal sequencing theorem in every iteration. Thus, we observe that all the algorithms are polynomial time bound, however, they differ in their complexities only in the actual magnitudes.

From the design of PPOLD and the heuristic strategies, it may be inferred that the choice of strategies purely depends on the size of the network (in the sense of number of processors) and the depth of the network (number of levels). Considering a sparse tree network with small depth, PPOLD and Strategy I may deliver more-or-less identical performance. For sparse networks, buffer availability and number of redundant processor-link pairs that are to be eliminated are two conflicting issues to be taken care in the design of strategies. In fact, Strategy I precisely attempts to do this. Although Rule A can bring an improvement in performance in general, because the network is sparse, amount of buffer space available with the original network becomes crucial. Hence, if Rule A in Strategy I eliminates a large number of processor-link pairs, then PPOLD becomes a better alternative to Strategy I. However, for dense networks, the conflicting nature of buffer availability and number of redundant processor-link pairs may not be significant and hence, Strategy I is expected to deliver a better performance. Similarly, when comparing strategies II and III, since all the processor-link pairs will be utilized, for sparse networks, Strategy II may be a better option. This is due to the fact that the gain in performance delivered by Strategy III will be very small when compared to its overhead, and hence, we can accept Strategy II. However, for dense networks, even with overheads, time performance delivered by Strategy III may be superior than Strategy II.

4.2. Extensions to arbitrary graphs

Although our primary objective is to design load distribution strategies for scheduling on arbitrary tree networks, the strategies proposed in Section 3 can be used for load distribution on arbitrary graphs. Thus, for arbitrary networks, one of the direct ways to schedule a load on a graph is by constructing a minimum cost spanning tree (MST) (Prim's algorithm, as an example), starting from the processor at which the load originates (root processor). The resulting tree, denoted as G_{MST}, spans all the processors from the root processor at which the load originates. It may be noted that, at this stage, we can adopt any standard known algorithms to generate MST, such as Kruskal's or Prim's algorithm [22]. It may be noted that using Fibonacci heaps, for an arbitrary graph $G = \langle N, E \rangle$ where N denotes the number of nodes interconnected via E communication links, Prim's algorithm can be run in time $O(E + N \log N)$[22]. Thus, depending on implementation style, the running time of these standard algorithms can be taken care. After this stage, from the given graph, the *costly* links are eliminated, that is those paths that incur the smallest communication delays between any pair of two processors are used. Having obtained G_{MST}, as a second step, we apply PPOLD and/or the above proposed strategies on this G_{MST}, starting from last level to the root, to compute the load distribution, depending on case (a) or (b), respectively. Note that this approach uses a uniport communication model as with the strategies proposed in this paper. However, it would be interesting to design load distribution strategies for arbitrary graphs employing multi-port model.

Moreover, although we can make use of any standard algorithm to generate G_{MST}, it should be noted that not all the standard algorithms may engender the best tree structure for distribution with finite-size buffer constraints. This is because during the construction of G_{MST}, the buffer capacity constraints imposed by the processors are seldom considered. Thus, rather than taking care of the buffer constraints at the scheduling level, an interesting extension of this problem could be to find a novel algorithm to construct G_{MST} from the graph taking account of the buffer constraints. Furthermore, it should be noted that regardless of how G_{MST} is constructed from a graph, PPOLD can still be employed to yield an efficient load distribution.

5. Conclusions

In this paper, we investigated the problem of scheduling a divisible load in an arbitrary, multi-level, unbalanced, tree network. Taking into account the communication delays and buffer constraints, we have designed a polynomial-time bound algorithm, referred to as PPOLD, to achieve the optimal processing time, when Rule A is satisfied for all the processor-link pairs in the network. The design of PPOLD is based on a systematic application of IBS at every iteration on a sub-tree. This is to determine an optimal buffer space that should be rendered for every iteration by a sub-tree. The design of PPOLD involves two phases, where the first phase is to determine an equivalent tree which is used in the second phase for actually distributing the load. This was designed exclusively for cases wherein Rule A is automatically satisfied.

We have then considered a case when Rule A is not satisfied by some of the processor-link pairs in the network, and for this case, we have proposed three heuristic strategies (Strategy I, II and III) to achieve a best possible solution. All the strategies are shown to have polynomial time complexities. Systematic design of these algorithms involves careful planning of buffer allocation in every iteration, identifying the best set of processor-link pairs that participate, followed by determination of an optimal load distribution. This is done by using the concepts of processor-link/buffer equivalence and application of IBS for every sub-tree in the network. Further, with PPOLD as a basis, for heuristic strategies II and III, the concept of optimal sequencing proposed in the literature is used. Although both the strategies (II and III) use optimal sequencing concept, the way in which it is applied in these strategies differs. Strategy III uses the optimal sequencing theorem dynamically, in every iteration, and hence, it is shown to be suitable for applications that involve processing divisible loads and that they are on time-critical missions. This would be a typical case while processing a large volume of divisible data for military surveillance applications, wherein timely delivery of the processed data is critical. In such situations, it is imperative to use an on-line version rather than Strategy II. Illustrative examples were given to demonstrate the complete workings of all the strategies proposed.

One of the most important issues that remain to be addressed at this juncture is to explore whether or not a single relationship be derived to identify all the redundant processor-link pairs with all the influencing parameters such as processor speeds, communication delays, load size and the buffer capacity constraints, respectively. Such an explicit relationship, without demanding the application of IBS algorithm and Rule A independently, is expected to be more efficient, if not optimal, to deliver a best performance. In fact, the relationship can then be applied to any arbitrary graphs on which loads are to be scheduled.

Appendix A

As a first step, for the purpose of continuity, we state the implications of Rule A now and then present the procedure to obtain an equivalent reduced single-level tree. Consider a single-level tree $\Sigma(x,i,m+1)$ shown in Fig. A1(a). It has been shown in Ref. [5] that to obtain an optimal solution for a load distribution problem in the case of single-level tree networks, as a first step, we need to eliminate redundant processor-link pairs. In order to eliminate such redundant processor-link pairs (slow processor-link pairs) from participating in the load computation process, we apply the following condition, referred to as Rule A, in the literature [5].

$$z_{i,j}T_{\mathrm{cm}} < z_{i,(j+1,\ldots j+h)}T_{\mathrm{cm}} + w_{i,(j+1,\ldots,j+h)}T_{\mathrm{cp}} \tag{A1}$$

where $z_{i,(j+1,\ldots,j+h)}$ is the equivalent link speed of links $l_{i,j+1}$ to $l_{i,j+h}$ and $w_{i,(j+1,\ldots,j+h)}$ is the equivalent processor speed of processors $p_{i,j+1}$ to $p_{i,j+h}$ in $\Sigma(x,i,m+1)$, respectively, as shown in Fig. A1(a) and as defined in Ref. [5]. The implications of the above expression can be easily understood in its simplest form, for $h=1$, which states the following.

$$z_{i,j}T_{\mathrm{cm}} < z_{i,(j+1)}T_{\mathrm{cm}} + w_{i,(j+1)}T_{\mathrm{cp}} \tag{A2}$$

Note that if the above condition is violated, then the time taken by the front-end of $p_{x,i}$ to communicate a load fraction to $p_{i,j}$ through $l_{i,j}$ is more than the time taken to communicate the same load through $l_{i,(j+1)}$ and process it at $p_{i,(j+1)}$. Hence, it is logical to send the load to $p_{i,(j+1)}$ than to $p_{i,j}$. The above condition formalizes this logic for a set of processors that can share the load of a redundant processor-link pair, thus eliminating it from the system. Thus, we systematically apply this Rule A, starting from the right hand side (processor-link pair $(p_{i,(m-1)}, l_{i,(m-1)})$) and obtain an optimal reduced tree $\Sigma^*(x,i,n+1)$, where $n \le m$,

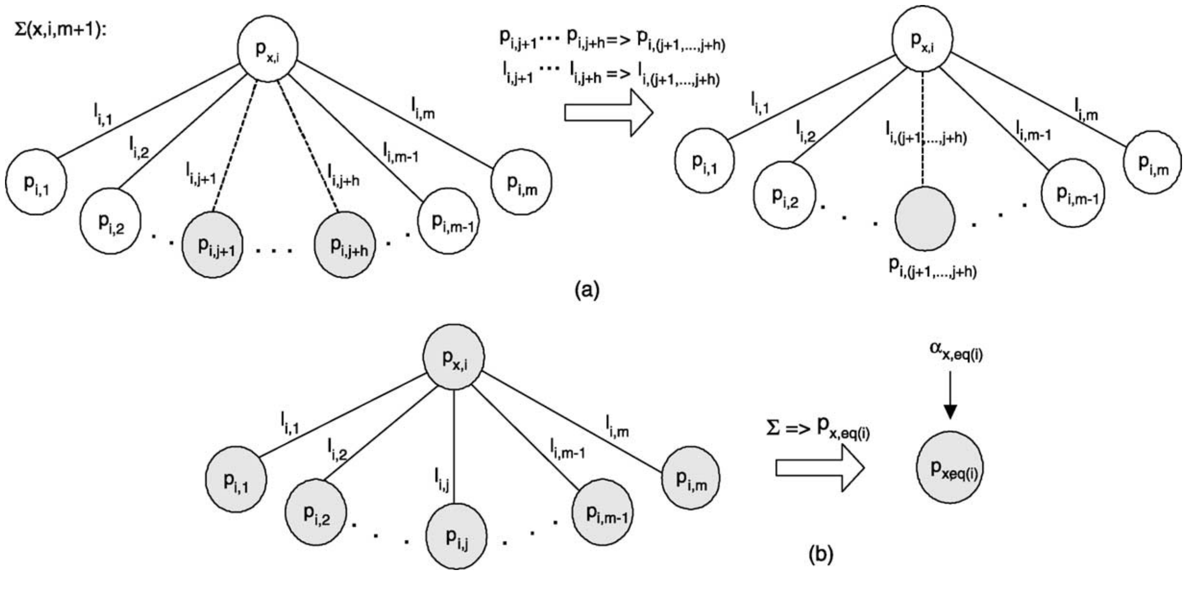

Fig. A1. Concepts of processor equivalence (a) equivalence of set of processor-link pairs used in Rule A; (b) equivalence of an entire single-level tree.

which is proven to be optimal in time performance in the literature. From Ref. [5], we obtain the expressions for $w_{i,(j+1,\ldots,j+h)}$ and $z_{i,(j+1,\ldots,j+h)}$ as

$$w_{i,(j+1,\ldots,j+h)} = \left(\frac{1}{1 + \sum_{u=j+2}^{j+h} \prod_{v=u}^{j+h} f_v} \right) w_{i,(j+h)} \tag{A3}$$

$$z_{i,(j+1,\ldots,j+h)} = \frac{\sum_{u=j+2}^{j+h} \left\{ \left(\prod_{v=u}^{j+h} f_v \right) z_{i,(u-1)} \right\} + z_{i,(j+h)}}{1 + \sum_{u=j+2}^{j+h} \prod_{v=u}^{j+h} f_v} \tag{A4}$$

where f_v is as defined in (3). Using the above concept of equivalent processor and equivalent link to the entire single-level tree network, we can obtain the processing time for the entire single-level tree network as

$$T(\Sigma(x,i,m+1)) = \alpha_{x,i} w_{x,i} T_{\mathrm{cp}}$$

$$= \left(\frac{\prod_{v=1}^{m} f_v}{1 + \sum_{u=1}^{m} \prod_{v=u}^{m} f_v} \right) w_{x,i} T_{\mathrm{cp}} = 1 \times w_{x,\mathrm{eq}(i)} \times T_{\mathrm{cp}} \tag{A5}$$

where

$$w_{x,\mathrm{eq}(i)} = \left(\frac{\prod_{v=1}^{m} f_v}{1 + \sum_{u=1}^{m} \prod_{v=u}^{m} f_v} \right) w_{x,i} \tag{A6}$$

Thus, given a single-level tree network, we can replace the entire network with a processor $p_{x,\mathrm{eq}(i)}$, as shown in Fig. A1(b), with its equivalent speed given by (A6). Thus, the optimal processing time is given by

$$T(\Sigma(x,i,m+1) = \alpha_{x,\mathrm{eq}(i)} \times L \times w_{x,\mathrm{eq}(i)} \times T_{\mathrm{cp}} \tag{A7}$$

and for $\Sigma(0,m+1)$, the optimal processing time is given by

$$T(\Sigma(0,m+1) = \alpha_{\mathrm{eq}(0)} \times L \times w_{\mathrm{eq}(0)} \times T_{\mathrm{cp}} = L \times w_{\mathrm{eq}(0)} \times T_{\mathrm{cp}} \tag{A8}$$

which is equal to the processing time of load L.

It may be noted that, Eq. (A8) is indeed the total processing time of the entire load on G_T.

References

[1] M.M. Eshaghian (Ed.), Heterogeneous Computing, Artech House, Norwood, MA, 1996.

[2] B. Wilkinson, M. Allen, Parallel Programming-Techniques and Applications using Networked Workstations and Parallel Computers, Prentice-Hall, Englewood Cliffs, NJ, 1999.

[3] L. Pangfeng, N.B. Sandeep, Experiences with parallel N-body simulation, IEEE Transactions on Parallel and Distributed Systems 11 (12) (December 2000) 1306–1323.

[4] Y.C. Cheng, T.G. Robertazzi, Distributed computation with communication delays, IEEE Transactions on Aerospace and Electronic Systems 24 (1988) 700–712.

[5] V. Bharadwaj, D. Ghose, V. Mani, T.G. Robertazzi, Scheduling Divisible Loads in Parallel and Distributed Systems, IEEE Computer Society Press, Los Almitos, California, 1996.

[6] M. Drozdowski, Selected problems of scheduling tasks in multiprocessor computer systems, Wydawnictwa Politechniki Poznanskiej (in English) Book No. 321, Poznan, Poland, 1997

[7] V. Bharadwaj, D. Ghose, T.G. Robertazzi, Divisible load theory: a new paradigm for load scheduling in distributed systems, http://opensource.nus.edu.sg/~elebv/DLT.htm

[8] J. Sohn, T.G. Robertazzi, Optimal divisible job load sharing on bus networks, IEEE Transactions on Aerospace and Electronic Systems 1 (1996).

[9] J. Blazewicz, M. Drozdowski, M. Markiewicz, Divisible task scheduling—concept and verification, Parallel Computing, vol. 25, Elsevier Science, Amsterdam, January 1999, pp. 87–98.

[10] G. Barlas, Collection-aware optimum sequencing of operations and closed-form solutions for the distribution of a divisible load on arbitrary processor trees, IEEE Transactions on Parallel and Distributed Systems 9 (5) (May 1998) 429–441.

[11] D. Ghose, H.J. Kim, Load partitioning and trade-off study for large matrix-vector computations in multicast bus networks with communication delays, Journal of Parallel and Distributed Computing (1998) 54.

[12] D.A.L. Piriyakumar, C.S.R. Murthy, Distributed computation for a hypercube network of sensor-driven processors with communication delays including setup time, IEEE Transactions on Systems, Man and Cybernetics—Part A: Systems and Humans 28 (2) (March 1998) 245–251.

[13] J. Sohn, T.G. Robertazzi, A multi-job load sharing strategy for divisible jobs on bus networks, CEAS Technical Report 665, State University of New York at Stony Brook, April 1993

[14] V. Bharadwaj, B. Gerassimos, Efficient scheduling strategies for processing multiple divisible loads on bus networks, Journal of Parallel and Distributed Computing 62 (2002) 132–151.

[15] V. Bharadwaj, H.F. Li, T. Radhakrishnan, Scheduling divisible loads in bus networks with arbitrary processor release times, Computer and Mathematics with Applications 32 (7) (1996).

[16] J. Blazewicz, M. Drozdowski, Distributed processing of divisible jobs with communication startup costs, Discrete Applied Mathematics 76 (1–3) (June 1997) 21–41.

[17] V. Bharadwaj, X. Li, C.C. Ko, On the influence of start-up costs in scheduling divisible loads on bus networks, IEEE Transactions on Parallel and Distributed Systems 11 (12) (2000) 1288–1305.

[18] X. Li, V. Bharadwaj, C.C. Ko, Scheduling divisible tasks on heterogeneous single-level tree networks with finite-size buffers, IEEE Transactions on Aerospace and Electronic Systems 37 (January 2001).

[19] V. Bharadwaj, S. Ranganath, Theoretical and experimental study on large size image processing applications using divisible load paradigm on distributed bus networks, Image and Vision Computing 20 (2002) 917–935.

[20] M. Drozdowski, P. Wolniewicz, Experiments with scheduling divisible tasks in clusters of workstations, Euro-Par 2000, LNCS 1900, Springer, Berlin, 2000, pp. 311–319.

[21] S.K. Chan, V. Bharadwaj, D. Ghose, Large Matrix-vector products on distributed bus networks with communication delays using the divisible load paradigm: performance analysis and simulation, Mathematics and Computers in Simulation 58 (2001) 71–92.

[22] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, MA, 1994.