Perfect load balancing on anonymous trees

Luciano Margara  Alessandro Pistocchi  Marco Vassura

October 2003

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

1. This work was partially supported by the Future & Emerging Technologies unit of the European Commission through Project BISON (IST-2001-38923).
Recent Titles from the UBLCS Technical Report Series

Perfect load balancing on anonymous trees

Luciano Margara\textsuperscript{3}  Alessandro Pistocchi\textsuperscript{4}  Marco Vassura\textsuperscript{5}


October 2003

Abstract
We study the problem of load balancing with finitely-divisible loads (also known as token distribution problem) on distributed anonymous tree networks. Existing results obtaining perfect load balancing assume prior knowledge of the number of nodes and an edge coloring of the tree [8]. In this paper we show that it is possible to obtain perfect load balancing on anonymous trees without further assumptions. Moreover, we are able to compute aggregate information on anonymous trees like the number of nodes and the total load. Our algorithm runs in $\Theta(M + n)$, where $M$ is the total number of tokens and $n$ is the number of nodes of the tree. To run our algorithm each node $v$ must be able to store $\Theta(d(\log M + \log n))$ bits of information, where $d$ is the degree of $v$. 

\textsuperscript{2} This work was partially supported by the Future & Emerging Technologies unit of the European Commission through Project BISON (IST-2001-38923).

\textsuperscript{3} Computer Science Department, University of Bologna. Email: margara@cs.unibo.it
\textsuperscript{4} Computer Science Department, University of Bologna. Email: pistocchi@cs.unibo.it
\textsuperscript{5} Computer Science Department, University of Bologna. Email: vassura@cs.unibo.it
1 Introduction.

A fundamental problem in distributed networks is to divide up workload effectively among processors. This type of problem is called load balancing. Load balancing problems have been studied under many different models of computations. We address the static load balancing problem on a single port model with bounded edge capacity, distributed control and finitely divisible load [4, 8]. The problem we study sometimes is referred to as Token distribution problem: each processor, called processing element (PE), of the distributed architecture is given an initial set of tokens. Each token represents a unit of load. The aim is to redistribute tokens so that final loads of PEs differ as little as possible. If such difference is at most one, we obtain a perfect load balancing. It is assumed that each token requires only a constant amount of time to be sent from one PE to an adjacent PE and that no token is created or destroyed before the redistribution process reaches a steady state. We assume synchronous single-port communication but, since all the algorithms we consider perform only local communications, each result can be easily ported to an asynchronous model by adding a constant number of local synchronization messages [1] (see Section 5.2 for the details). We assume bi-directional communication links but note that they can be simulated by a constant number of communication steps over uni-directional links. As usual, we model a network by an undirected graph where nodes represent PEs and edges represent communication links. Nodes are anonymous. This means that nodes do not have identity numbers: our algorithm requires no information about nodes identities.

Previous work on Token Distribution. For general networks Ghosh et al. [4] analyse two algorithms which reduce the minimum difference in tokens between any two nodes to at most $O(d^2 \log n)/\alpha$, where $d$ is the maximum degree of any node in the network, $n$ is the number of nodes in the network, and $\alpha$ is the edge expansion of the network. Many researchers studied token distribution on specific topologies. For ring networks Gehrke, Plaxton, and Rajaraman [3] present and analyse an algorithm which converge to a perfect balanced state with message complexity asymptotically optimal. For meshes and torus Houle et al. [7] proposed and analysed an algorithm reducing the discrepancy to the minimum degree of the net in worst-case optimal time. The same algorithm used for complete binary trees obtain a discrepancy which is at most the height of the tree [6]. For arbitrary trees Houle, Symvonis, and Wood [8] give an algorithm which, assuming an edge coloring and a global knowledge of the number of nodes in the tree, reaches perfect load balancing. Algorithms computing global network information can be found in [5, 10, 2] but in a very different model, without anonymity but considering fault tolerance issues.

Our results. We present an algorithm for solving the token distribution problem on anonymous trees (solving a problem left open in [8]). To our knowledge this is the first algorithm that computes a perfect distribution of tokens without receiving as input additional information on the network such as the total number of nodes or the total number of tokens.

We get a perfect load balancing in $\Theta(M + n)$ steps, where $M$ is the number of tokens and $n$ the number of nodes in the tree.

Our technique is based on a three phase approach. In Phase 1 we compute number of nodes and of tokens in the tree in $\Theta(n)$ steps. The same algorithm can be used to compute many other global information on the tree, provided that it can be computed from information known at single nodes (Section 5.3). In multi-port models such information is computed in $\Theta(diam)$, where $diam$ is the diameter of the tree. In Phase 2 load is balanced using information in Phase 1: knowing number of nodes and load of each subtree, tokens are sent to the subtree with the minimum average number of tokens. After Phase 2 we reach a balancing with maximum discrepancy $n - (M \mod n)$. Such balancing is refined in Phase 3 collecting extra tokens from nodes with above average load and then redistributing them to subtrees with under average load. At the end of this Phase we get a perfect load balancing (Figure 1).
2 Basic definitions.

For a better comprehension of the algorithm and to simplify the correctness proof we introduce a new inductive definition of tree.

**Definition 2.1.** Let \( G = (V, E) \) be a connected graph. A leaf of \( G \) is a vertex \( v \in V \) such that \( \deg(v) = 1 \). Let \( Le(G) \subseteq V \) be the subset of leaves of \( G \). \( G \) is a recursive tree if and only if:

- Base Case 1: \( V = \{v\}, E = \emptyset \)
- Base Case 2: \( V = \{v_1, v_2\}, E = \{(v_1, v_2)\} \)
- Induction Step: \( V = Le(G) \cup V' \), with both \( Le(G) \) and \( V' \) non void, and the subgraph of \( G \) induced by \( V' \) is a recursive tree.

In what follows we will refer to the nodes of the two base cases respectively with root and central nodes.

**Lemma 2.2.** A connected graph \( G \) is a recursive tree if and only if it is a tree.

Proof of this lemma is in Appendix A.

3 A \( \Theta(M + n) \) perfect load balancing on anonymous trees.

Our algorithm operates in three phases: first it collects information, then uses it to balance and finally refines the balancing to be perfect. We require that each node can store \( \Theta(d \log M + \log n) \) bits of information and that it can recognize communication links to different neighbors.

We give (Algorithm 1) pseudo code of the algorithm running at each node, called Tree-Balance, which will be further expanded and detailed in Appendix B.

We show an example of computation in Figure 1: each node contains an initial amount of tokens, arrows indicate a flow of information or tokens. The idea is that each node \( v \) collects the total load and number of nodes in the subtree rooted at \( v \) and send such information up on the tree (Fig. 1 A). In this way the root, or the two central nodes, will receive information about all the tree, calculate the total load and number of nodes and distribute them on the tree. After that, called Phase 1, every node knows the total load and number of nodes in each subtree and in the whole tree, so it can compute the average load and send its extra token to needing subtrees. This is called Phase 2, after which we reach a good balancing (Fig. 1 B). To reach perfect balancing, in Phase 3, nodes with more than average load send extra tokens to central nodes (Fig. 1 C) and then central nodes redistribute them where needed (Fig. 1 D, Section 4).

4 Correctness proof.

In this Section we prove that our algorithm effectively computes number of nodes and tokens and reaches a perfect load balancing. Lines numbers refer to the detailed pseudo code of Appendix B.

First we use the recursive tree defined in Section 2 to give an inductive proof that Phase 1 of Tree-Balance algorithm works on any tree. We will refer to total load as \( Tload \) and total number of nodes as \( Tnodes \), as in the algorithm.

**Theorem 4.1.** Let \( G = (V, E) \) be a tree synchronous network with \( V = \{v_1, ..., v_n\} \). Let \( v \in V \) be any node of \( G \) and let \( V_v \) be the neighborhood of \( v \). Then Phase 1 of Tree-Balance terminates in \( \Theta(|V|) \) steps and is correct (i.e. at the end every node knows \( Tnodes \) and \( Tload \)).

**Proof.** Suppose \( G = (\{v\}, \emptyset) \); Then \( l = \emptyset \) and \( V_v = \emptyset \). Then no message is sent or received and \( v \) \( Tnodes \) and \( Tload \) get the right values (lines 27-28) during first time step. Suppose \( V = \{v_1, v_2\}, E = \{(v_1, v_2)\} \); then during the first time step \( v_1 \) and \( v_2 \) send to each other their value (lines 13-17), during the second one they receive them (lines 18-25) and save global values just computed (lines 26-30). One more time step will be used to exchange such global values (lines 39-42), total time steps \( 2 + 1 = 2 \times \Theta(2) = \Theta(2) \). So the theorem holds for both base cases.
Algorithm 1 Tree-Balance
executed at each node \( v \)

**PHASE 1: collect information**
wait for \( subTree_v \) info from neighbors
if \( subTree_v \) info received from all neighbors but one then
  send own \( subTree_v \) info to that neighbor
  wait for \( Tree \) info from that neighbor
  compute \( subTree_v \) info for that neighbor
  forward \( Tree \) info to other neighbors
end if
if \( subTree_v \) info received from all neighbors then
  compute \( Tree \) info
  send \( Tree \) info to all neighbors
end if
wait for synchronization

**PHASE 2: first balancing**
compute average load \( av \)
for \( totalload \) times do
  if \( ownLoad_v > av + 1 \) then
    send a token to subtree with less average tokens
  end if
end for

**PHASE 3: refine balancing**
for \( (totalnodes)/2 \) times do
  if \( ownLoad_v > av \) and \( v \) is not a central node then
    send a token towards a central node
  end if
end for
for \( 1.5(totalnodes) \) times do
  if \( ownLoad_v > av + 1 \) then
    send a token to subtree with less average tokens
  else if \( ownLoad_v > av \) then
    send a token to subtree with less average tokens only if it has less than \( av \) tokens
  end if
end for
Figure 1. Tree-balancing algorithm: (A) Phase 1: Local information sent to central node, tokens do not move. (B) End of Phase 2: token redistributed to subtrees with fewer tokens, first balancing. (C) Phase 3: Extra token collected in central node. (D) End of Phase 3: Extra token redistributed, perfect load balancing.
Now suppose \( G = Le(G) \cup V' \), with both \( Le(G) \) and \( V' \) non void. Then at the first time step we have each node \( v \in Le(G) \) sending its values to the father (lines 13-17) and nothing else happens. At second step each father receives messages sent in first step and updates its local variables (lines 18-25). If there is a node which received values from all its neighbors our tree is a star and such node is the root: the root will communicate to all leaf the global information computed and, after \( n - 1 \) steps, each leaf will receive them. Otherwise at third step each node in \( V' \) enters while loop (line 12) as if the original tree was \( G' = (V', E') \) with load at each leaf as the sum of the load of its original load plus the load of each son in \( G \) and cardinality of each leaf equal to its number of sons in \( G \) plus 1. Applying induction hypothesis in \( \Theta(|V'|) \) steps every node in \( V' \) will know the total load and the total number of nodes of \( G' \). Since the load and node cardinality of each leaf of \( G' \) was that of the corresponding subtree in \( G \) such values are equals to the total load and total number of nodes of \( G \). Thanks to line 38-48 in at most further \( |Le(G)| \) steps also nodes in \( Le(G) \) get the same information: the algorithm is correct and runs in at most \( \Theta(|V'|) + |Le(G)| = \Theta(|V'|) \) time steps.

As shown in theorem 4.1 Phase 1 of our algorithm runs in time linear in the number of nodes. It is easy to see that this is due to the single-port model that when distributing the computed global values force, in worst case, to communicate to one node at each step. With little changes the proof of theorem 4.1 can show that for multi-port model Phase 1 runs in time linear in the diameter of the tree. This can be useful if extending the algorithm to compute other aggregate function of the tree (Section 5.3).

**Theorem 4.2.** After Phase 2 all nodes have \( load_v \leq av + 1 \).

**Proof.** Since all tokens sent go toward regions of the tree with less tokens, since at each step with at least one node with more than \( av + 1 \) tokens at least one token is sent and since the total number of token is less than \( (av + 1)/Tnodes \), after a certain number of step tokens will be distributed on the whole tree granting that each node has less than \( av + 1 \) tokens. It easy to see that the worst case, respect to the number of steps required to the distribution, is the one where all tokens are initially at one node. This worst case requires \( Tload \) time steps, so our first while loop, iterating for \( Tload \) time steps, makes all nodes have \( load_v \leq av + 1 \).

We already get a good balancing, as shown by the following corollary.

**Corollary 4.3.** After Phase 2 the maximum discrepancy possible is \( disc = Tnodes - (Tload \mod Tnodes) \).

**Proof.** Since for theorem 4.2 all nodes have \( load_v \leq av + 1 \) the most unbalanced case is one node with as few Tokens as possible and all other nodes with \( av + 1 \) Tokens. The least number of Token possible is then \( Tload - (Tnodes - 1)(av + 1) = (Tnodes \cdot av + (Tload \mod Tnodes)) - Tnodes \cdot av - Tnodes + av + 1 = (Tload \mod Tnodes) - Tnodes + av + 1 \). So the maximum discrepancy possible is \( (av + 1) - ((Tload \mod Tnodes) - Tnodes + av + 1) = Tnodes - (Tload \mod Tnodes) \).

To reach discrepancy 1 we need to redistribute Tokens above average, this is done in two loops of Phase 3: the first accumulate them at the center of the tree, the second distribute them as needed.

**Theorem 4.4.** After Phase 3 first loop (lines 2-14) there are at most two nodes with \( load_v > av \) and are in a central position of the tree.

**Proof.** During such loop nodes send all tokens above \( av \) in the ‘up’ direction. The only nodes which do not send their tokens are the ones which computed the number of nodes in Phase 1. As shown in proof of theorem 4.1 such nodes are not in \( Le(G) \), they are not leaves. If \( V' \) can be divided in two nonempty sets \( Le(V') \) and \( V'' \) for the same reason they can not be in \( Le(V') \). Iterating this reasoning we get that such nodes must be one of the base cases of the recursive tree: one or two nodes. Such nodes, by construction of the recursive tree, are at the center of the tree: the subtrees rooted at their sons have about the same number of nodes. The time needed to the other nodes to send their extra token is at most the maximum distance from leafs to the center. This is half the diameter of the tree, so our second loop does not need to iterate more than half the number of nodes time steps.
Corollary 4.5. After Phase 3 first loop (lines 2-14) the maximum load possible at a node is \( av + disc + (Tload \mod Tnodes) = av + Tnodes \).

Proof. Since, for theorem 4.4, all nodes but the central ones have \( load < av \) the worst case is with all nodes with the as less Token as possible and one with all other Tokens. During this loop, however, the only nodes sending Tokens are the ones with \( load_w > av \), so all other nodes do not decrease their \( load_w \) and no node lowers its \( load_w \) under \( av \). So, for corollary 4.3, the least number of Tokens a node can have is \( av + 1 - disc \) and if there is more than one node with \( load_w < av \) the total discrepancy is less than \( disc \). This means that at the central node we can accumulate at most \( Tload - (Tnodes - 1)av + disc = av + (Tload - Tnodes \cdot av) + disc = av + disc + (Tload \mod Tnodes) = av + Tnodes \) Tokens.

Theorem 4.6. After Phase 3 second loop (lines 17-34) all nodes have \( av + 1 \geq load_w \geq av \).

Proof. For theorem 4.4 all nodes except at most 2 have \( load_w \leq av \) and such 2 nodes are at the center of the tree. So Token distribution starts from center nodes and Tokens, going in minimum load/nodes subtrees, go toward the leaves where they are most needed. Eventually the ratio load/nodes will be greater than \( av \) for each subtree: if this is not the case then at least one subtree has such ratio lesser than \( av \) and, due to the limited number of Tokens, at least one node has \( load \geq av \) and is near a subtree containing the one with low ratio, such node will give its token above \( av \) to the subtree (lines 22-26) and such reasoning can be repeated until the subtree has \( av \) ratio. All nodes with less than \( av \) Tokens will receive enough tokens because, since their parents send to them all Tokens above \( av \), the load/nodes ratio of their subtree will not reach \( av \) until they have at least \( av \) Tokens. To understand this remember that each subtree, except for the center-rooted ones, has initially less than \( av \) Token at each node and that all Tokens come from the center. For corollary 4.5, the number of tokens each node distribute is at most \( Tnodes - 1 \) and, as said, they all start from center to leaves, so in at most \( Tnodes - 1 \) steps all nodes have \( load_w \leq av + 1 \). To reach such state the loop needs to iterate at most for the maximum distance from leaf to center, plus the maximum load to be transferred, time steps. For corollary 4.5 and for graph properties these are bounded by \( Tnodes + Tnodes/2 \) which are the number of steps iterated by our loop.

4.1 Space and time complexity.

We call \( d \) the maximum degree of the tree, \( M \) the total load, \( diam \) the diameter of the tree and \( n \) the number of nodes. As proved in Section 4 Phase 1 runs in (worst case) \( \Theta(n) \) in single-port model and \( \Theta(diam) \) in multi-port model. Since we address single-port model the algorithm synchronizes making each node enter the Phase 2 after \( n \) steps. Phase 2 uses \( M \) steps, while Phase 3 uses \( n/2 + n + n/2 + 1 \) steps, which is \( \Theta(n) \). The total time complexity is then \( \Theta(M + n) \).

As said in proof of theorem 4.4 the real bound for loops of Phase 3 (lines 2-14 and 17-34) is \( diam/2 \) instead of \( n/2 \), so a natural and simple optimization is to compute \( diam \) in Phase 1 and use it in Phase 3. To compute \( diam \) in Phase 1 we have to just add \( m \) to messages \( subTload_w \cdot subTnodes \), (line 15), compute the maximum \( m \) in subtrees (add it between lines 19 and 24) and redistribute the result (add \( diam \) to messages of lines 40 and 45). Time and space complexity for Phase 1 are increased only by a constant amount, while substituting \( diam/2 \) to \( Tnodes/2 \) in loops of the Phase 3 we get a \( M + diam + n + diam + 1 \) steps for Phase 3. This optimization surely increases medium case performance of the algorithm but worst case (tree is a line) asymptotic complexity remains \( \Theta(M + n) \).

The space needed to store algorithm data at each node is that for subtrees load and number of nodes, plus a constant number of local data of size less or equal to the elements of subtrees array, so size complexity at single node is \( \Theta(d \log M + \log n) \).

5 Extensions.

The algorithm we presented in Section 3 is in its simpler and easier to understand version. In this Section we show how to extend it to work on generic networks, on asynchronous model, and to
5.1 General networks.

To extend our algorithm to work on any network, instead of trees, we want to add an additional Phase 0 which builds a spanning tree of the original network. The problem is that we have a distributed anonymous network. So we can not use centralized procedures, nor we can use typical distributed algorithms, because they assume to know node identities. Furthermore it is known that no deterministic distributed algorithm can find a spanning tree in an anonymous network[11, 9]. For this reason it is impossible to find a straightforward extension to our algorithm for generic networks because, as seen during the correctness proof, we would implicitly build a spanning tree. The only way is then to relax our anonymity condition or use approximated algorithm. Without giving a unique identifier to each node, or elect a leader, we can, as example, use an algorithm equal at each node except one, as in [9]. Approximated algorithm give good probabilities to find a spanning tree but they may generate infinite computations.

5.2 Asynchronous model.

Assume FIFO communication links and bounded message delays (messages eventually arrive). The simpler way to make our algorithm work asynchronously is to map each synchronous message to an asynchronous sequence of ack and wait of messages, as explained in[1]. This would ensure that Tree-Balance works on the asynchronous model without modifications\(^6\), because we only do local communications without really using the global synchrony of the network, but could be very slow. We can do better working directly on Tree-Balance. It is easy to understand that the Phase 1 works also on asynchronous trees without modifications, the only difference being which nodes computing the global values for load and number of nodes (obviously variable step has no more meaning and nodes do not terminate all at the same time). Due to single-port and local communications also the Phase 2 and 3 work rather well asynchronously but there are scenarios where not all nodes stops or Token are lost. As example if one process terminates and the central node redistributes token to it such token are lost. As example of non termination consider the case where the central node, during Phase 3, gives token to a node which has av token and is still in Phase 2: the process will give back the token, then the central node will give it to the process again and so on. We then need to synchronize nodes in order to make them execute the same loops at the same time: we do it adding special synchronization messages at the end of each loop, so that every node waits to receive \(T_{nodes} - 1\) such messages before continuing computation.

5.3 General graph properties.

In Phase 1 our algorithm collects and redistributes global information about the tree, such as the number of nodes, using only local communication. The same algorithm can be used to compute any global function of the tree, provided that it does not change locally during computation.

**Observation 5.1.** Let \(G = (V, E)\) be a tree with \(V = \{v_1, \ldots, v_n\}\). Let \(f : V \to \mathbb{R}\) be a function, \(\mathbb{R}\) being a set on which it is defined a commutative and associative binary operator \(\bullet\) and being closed respect to it. Then the Phase 1 of our algorithm can be easily modified so that it computes \(\hat{f}(G) = f(v_1) \bullet \ldots \bullet f(v_n)\) and terminates in \(\Theta(diam(G))\) steps and is correct (i.e. at the end every node knows \(\hat{f}(G)\)).

**Proof.** The proof follows from a trivial modification of proof of Theorem 4.1. \(\square\)

As example we could compute the total amount of free storage on the network, even asynchronous networks (see Section 5.2), or the number of free terminals, or the list of files stored (shared) in all processors. Note that if we do not need particular computation on subtree values we do not need to store partial information, so each node must be able to store just the global information it needs to know.

---

\(^6\) Except for those of footnote 7
6 Conclusions and further work.

We have shown a linear time distributed algorithm for computing perfect balancing on anonymous trees which works on both synchronous and asynchronous models. It can be adjusted to compute any other global information of the tree. Moreover, we explained why deterministic extension to general network is unfeasible, unless we relax anonymity constraint. In doing that we obtain a perfect load balancing distributed algorithm on general networks running in linear time plus the cost of building a spanning tree.

Further research consist of extending the algorithm to dynamic models, where the number of token at each node changes during time. It is also interesting to extend the algorithm to compute and maintain aggregates (global information) in Peer to Peer or Scale Free networks.

A Proof of Lemma 2.2.

Lemma 2.2. A connected graph $G$ is a recursive tree if and only if it is a tree.

Proof. Let $G$ be a recursive tree. It is obvious that both the two base cases are trees. So suppose $V = Le(G) \cup V'$, $Le(G) \neq \emptyset, V' \neq \emptyset$ and $G$ has a cycle. None of the nodes in $Le(G)$ can be part of the cycle, since they have degree 1, so that the cycle nodes must be contained in $V'$. But by applying the inductive hypothesis, $V'$ cannot have cycles, so $G$ also cannot. Then $G$ is a tree.

Let $G$ be a tree. If $|V| = 1$ then the only possible tree is our base case 1; similarly, if $|V| = 2$ then the only possible tree is our base case 2. If $|V| > 2$, then $V = Le(G) \cup V'$, $Le(G) \neq \emptyset, V' \neq \emptyset$ and the subgraph induced by $V'$ is connected, because removing leaves cannot alter connectedness, and so it is a tree itself. Indeed, for each leaf removed we remove also an edge, so that, defining $E'$ as the set of edges of the subgraph induced by $V'$, $|E'| = |V'| - 1$ so that the subgraph induced by $V'$ is a tree. By using the inductive hypothesis, the subgraph of $G$ induced by $V'$ is a recursive tree and then also $G$ is a recursive tree.

B Implementation issues.

In this Appendix we give pseudo code for each of the three phases and describe them in full details. This pseudo-code is useful for reference during the correctness proof (see Section 4). We assume synchronous communication, to simplify the pseudo-code assume that each loop cycle corresponds to a time step and that all communications start at the end of the loop cycle and are resolved before the beginning of the new one. See Section 5.2 to change the algorithm for asynchronous models.

PHASE 1

1: let $m$ be the number of neighbors of $v$
2: shoot $\leftarrow m$
3: shot $\leftarrow -1$
4: done $\leftarrow 0$
5: label arcs adjacent to $v$ from 0 to $m - 1$
6: load array of $m$ integers initially −1
7: node$e$s array of $m$ integers initially −1
8: load$e$v $\leftarrow$ initial load at $v$
9: subTload$e$v $\leftarrow$ load$e$v
10: subTnode$e$s$e$v $\leftarrow$ 1
11: step $\leftarrow 0$
12: while done $= 0$ do
13: \hspace{1em} if shoot $= 1$ and shot $= -1$ then

7. To drop this assumption would suffice to rewrite each if-end if and for-endfor block handling messages as parallel procedures
14: \(i \leftarrow \text{arc for which } load[i] = -1\)
15: send \(\text{subTreeLoad}_i; \text{subTreeNodes}_v\) through \(i\)
16: \(shot \leftarrow i\)
17: \(\text{end if}\)
18: \(\text{for all messages } \text{subTreeLoad}_u; \text{subTreeNodes}_u\) received \(\text{do}\)
19: let \(i\) be the arc through which came the message
20: \(load[i] \leftarrow \text{subTreeLoad}_u\)
21: \(\text{nodes}[i] \leftarrow \text{subTreeNodes}_u\)
22: \(\text{subTreeNodes}_v \leftarrow \text{subTreeNodes}_v + \text{subTreeNodes}_u\)
23: \(\text{subTreeLoad}_v \leftarrow \text{subTreeLoad}_v + \text{subTreeLoad}_u\)
24: \(shot \leftarrow shot - 1\)
25: \(\text{end for}\)
26: if \(shot = 0\) then
27: \(\text{Tnodes} \leftarrow \text{subTreeNodes}_v\)
28: \(\text{Tload} \leftarrow \text{subTreeLoad}_v\)
29: \(\text{done} \leftarrow 1; shot \leftarrow -2\)
30: \(\text{end if}\)
31: if message \(\text{Tnodes}; \text{Tload}\) received through arc \(i\) then
32: \(load[i] \leftarrow \text{Tload} - \text{subTreeLoad}_v\)
33: \(\text{nodes}[i] \leftarrow \text{Tnodes} - \text{subTreeNodes}_v\)
34: \(\text{done} \leftarrow 2\)
35: \(\text{end if}\)
36: \(step \leftarrow step + 1\)
37: \(\text{end while}\)
38: if \(\text{done} = 1\) then
39: \(\text{for } i = 0 \text{ to } m - 1 \text{ do}\)
40: send \(\text{Tnodes}; \text{Tload}\) through arc \(i\)
41: \(step \leftarrow step + 1\)
42: \(\text{end for}\)
43: \(\text{else}\)
44: \(\text{for all arcs } j \text{ adjacent to } v \text{ except } i \text{ do}\)
45: send \(\text{Tnodes}; \text{Tload}\) through arc \(j\)
46: \(step \leftarrow step + 1\)
47: \(\text{end for}\)
48: \(\text{end if}\)
49: \(\text{while } step < \text{Tnodes} + 1 \text{ do}\)
50: \(step \leftarrow step + 1\)
51: \(\text{end while}\)

First data structures are initialized (lines 1-11): \(shot\) contains the number of arcs from which subtree information are still to be received, \(shot\) is a flag which becomes true when the node has sent its subtree information and is needed in second phase to give the 'up' direction, arrays \(load\) and \(\text{nodes}\) contain respectively the total load and number of nodes of subtrees adjacent to \(v\), \(\text{done}\) is a flag that becomes positive when \(v\) has retrieved all information, \(step\) counts time steps and is needed for synchronization, \(\text{load}_v\) contains the initial load at \(v\), \(\text{subTreeLoad}_v\) and \(\text{subTreeNodes}_v\) contain respectively the total load and number of nodes of subtree rooted in \(v\).

During this phase nodes collect information. If \(v\) is a leaf or if it has already collected information from all descending subtrees it sends such information (load and number of nodes of subtree rooted at \(v\)) up on the tree (lines 13-17): \(shot = 1\) identifies this situation and the 'up' direction is the one from which subtree information was not collected, and so the only one for which \(load[i] = -1\). Subtrees information are retrieved thanks to messages sent by other nodes (lines 18-25), such information are stored to be used in next phases (lines 20-21) and summed to compute information for the subtree rooted at \(v\) (lines 22-23). If \(v\) is the root of the tree (or one of the central nodes) it will receive subtree information from all its neighbors (lines 26-30), so it will compute
total load and number of nodes (lines 27-28) and send them to all its neighbors (lines 39-42). If some other node computes the total number of nodes and load, \( v \) will receive them (lines 31-35), then it will save the number of nodes and load on the subtree from which it received information (lines 32-33) and forward the message received to the other neighbors (lines 44-47). The while loop ends when \( v \) knows total load and nodes of all neighbors-rooted subtrees and of the whole tree. In lines 38-48 \( v \) sends to its neighbors this last information. At the end of first phase each node waits \( T_{\text{nodes}} + 1 \) time step (lines 49-51), for synchronization (see Section 5.2).

**PHASE 2**

1: \( av \leftarrow \lfloor T_{\text{load}}/T_{\text{nodes}} \rfloor \)
2: \( step \leftarrow 0 \)
3: \( j \leftarrow \text{arc for which } load[j]/\text{nodes}[j] \text{ is minimum} \)
4: \( \text{while } step < T_{\text{load}} \text{ do} \)
5: \( \quad \text{if } load_v > av + 1 \text{ then} \)
6: \( \quad \quad \text{send one Token through arc } j \)
7: \( \quad \quad load_v \leftarrow load_v - 1 \)
8: \( \quad \quad load[j] \leftarrow load[j] + 1 \)
9: \( \quad \text{end if} \)
10: \( \quad \text{for all Token received do} \)
11: \( \quad \quad i \leftarrow \text{arc through which came the Token} \)
12: \( \quad \quad load_v \leftarrow load_v + 1 \)
13: \( \quad \quad load[i] \leftarrow load[i] - 1 \)
14: \( \quad \text{end for} \)
15: \( \quad j \leftarrow \text{arc for which } load[j]/\text{nodes}[j] \text{ is minimum} \)
16: \( \quad step \leftarrow step + 1 \)
17: \( \text{end while} \)

In second and third phase the actual load balancing is done. First initialization (lines 1-3): \( av \) is the average load, \( step \) counts time steps and is used for synchronization, \( j \) is the arc linking to the neighbor-routed subtree with the least ratio load/nodes. In balancing loop (lines 4-17) if node \( v \) has more than \( av + 1 \) Tokens it sends one Token to the subtree with the least ratio \( load/\text{nodes} \) (lines 5-9), updating load information accordingly (lines 17-18). If neighbors of \( v \) sends token to him the load is updated (lines 10-14). At each loop (time step) \( j \) and \( step \) are updated (lines 15-16). The loop finishes after \( T_{\text{load}} \) time steps.

**PHASE 3**

1: \( step \leftarrow 0 \)
2: \( \text{while } step < T_{\text{nodes}}/2 + 1 \text{ do} \)
3: \( \quad \text{if } load_v > av \text{ and } shot \geq 0 \text{ then} \)
4: \( \quad \quad \text{send one Token through arc } shot \)
5: \( \quad \quad load_v \leftarrow load_v - 1 \)
6: \( \quad \quad load[shot] \leftarrow load[shot] + 1 \)
7: \( \quad \text{end if} \)
8: \( \quad \text{for all Token received do} \)
9: \( \quad \quad i \leftarrow \text{arc through which came the Token} \)
10: \( \quad \quad load_v \leftarrow load_v + 1 \)
11: \( \quad \quad load[i] \leftarrow load[i] - 1 \)
12: \( \quad \text{end for} \)
13: \( \quad step \leftarrow step + 1 \)
14: \( \text{end while} \)
15: \( step \leftarrow 0 \)
16: \( j \leftarrow \text{arc for which } load[j]/\text{nodes}[j] \text{ is minimum} \)
17: \( \text{while } step < T_{\text{nodes}} + T_{\text{nodes}}/2 \text{ do} \)
18: \( \quad \text{if } load_v > av + 1 \text{ then} \)
send one Token through arc \( j \)

\[ \text{load}_v \leftarrow \text{load}_v - 1 \]

\[ \text{load}[j] \leftarrow \text{load}[j] + 1 \]

**else if** \( \text{load}_v > av \) **and** \( \text{load}[j]/\text{nodes}[j] < av \) **then**

send one Token through arc \( j \)

\[ \text{load}_v \leftarrow \text{load}_v - 1 \]

\[ \text{load}[j] \leftarrow \text{load}[j] + 1 \]

**end if**

for all Token received do

\( i \leftarrow \text{arc through which came the Token} \)

\[ \text{load}_v \leftarrow \text{load}_v + 1 \]

\[ \text{load}[i] \leftarrow \text{load}[i] - 1 \]

**end for**

\( j \leftarrow \text{arc for which } \text{load}[j]/\text{nodes}[j] \text{ is minimum} \)

\[ \text{step} \leftarrow \text{step} + 1 \]

**end while**

After Phase 2 each node has no more than \( av + 1 \) Tokens but it could have less than \( av \) Token. So each node with more than \( av \) Tokens, except the central nodes, sends the extra Token ‘up’ to the central nodes (lines 2-14). After \( T/2 \) time steps the only node with more than \( av \) Tokens are the central nodes and the loop stops. Such Tokens are redistributed towards subtrees with least ratio load/node (lines 17-34). In such loop if some node has less then \( av \) Tokens its subtree has ratio load/nodes under \( av \), so if \( v \) has at least \( av \) Token sends all other Tokens to them (lines 22-26). Again these loops terminate after the maximum number of step required to do such operations (lines 2 and 17), see Section 4 for a proof that balancing is reached.

### References


