Perfect Token Distribution on Trees*

Luciano Margara[1], Alessandro Pistocchi[2], and Marco Vassura[3]

[1] Computer Science Department, University of Bologna.
margara@cs.unibo.it
[2] Computer Science Department, University of Bologna.
pistocch@cs.unibo.it
[3] Computer Science Department, University of Bologna.
vassura@cs.unibo.it

Abstract. Load balancing on a multi-processor system consists of redistributing tasks among processors so that all processors end up with roughly the same amount of work to perform. The *token distribution problem* is a variant of the load balancing problem where each task has unit-size and it represents an *atomic* element of work. We present an algorithm for computing a perfect token distribution (each processor has either $\lceil T/N \rceil$ or $\lfloor T/N \rfloor$ tasks, where N is the number of processors and T is the number of tasks scattered among processors) on distributed tree-connected networks having worst-case running time $O(TD)$ (D denotes the diameter of the tree). The number of token exchanges exceeds the optimum by at most $O(D \min\{T, N\})$.
In order to compute a perfect token distribution each node v must be able to store $\Theta(d_v(\log T + \log N))$ bits, where d_v is the degree (number of adjacent nodes) of v. This is the first fully decentralized algorithm for computing perfect token distributions on arbitrary tree-connected networks which does not receive as input any kind of aggregate information about the network (e.g., number of nodes or total number of tokens).

1 Introduction.

The performance of a distributed network crucially depends on dividing up work effectively among its processing elements [8]. This type of *load balancing* problem has been studied in many different models. The basic idea in all of the models is to evenly redistribute initial job load among processors (static balancing) and to keep load distribution as balanced as possible during time (dynamic balancing). Many variants of the load balancing problem have been proposed and widely investigated in the literature [1,2,3,4,6,9,10,11].

In this paper we consider a basic variant of the static load balancing problem, called *token distribution* problem, restricted to tree-connected networks. A tree-connected network is represented by a tree with N nodes. Each node represents

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

a processing element of the network and possesses a number of jobs of unit-size (tokens) to be processed. The total number of tokens is denoted by T. In a single message a token can be moved from any node to any other adjacent node in the tree. No token is created or destroyed during the redistribution process. The goal is to redistribute tokens across the tree so that each node ends up (being aware of that) with either $\lceil T/N \rceil$ or $\lfloor T/N \rfloor$ tokens (*perfect* token distribution). We adopt asynchronous single-port communication model with uni-directional communication links. Nodes are *anonymous*, i.e. they do not have identification labels. Synchronization between nodes (executing the same code in parallel) is achieved by exchanging messages.

Previous results. For general networks Ghosh et al. [4] analyze two algorithms which reduce the maximum difference in tokens between any two nodes (called "discrepancy") to at most $O((d^2 \log N)/\alpha)$, where d is the maximum degree of the nodes of the network, N is the number of nodes in the network, and α is the edge expansion of the network. Many results have been carried out on specific network topologies. For ring networks Gehrke, Plaxton, and Rajaraman [3] give an algorithm having an asymptotically optimal message complexity which converges to a perfectly balanced state.
For meshes and torus Houle et al. [7] give an algorithm that reduces the discrepancy to the minimum degree of the nodes of the network and that runs in worst-case optimal time. The same algorithm used for complete binary trees obtains in the worst case a discrepancy equal to the height of the tree [6].
For arbitrary trees Houle, Symvonis, and Wood [5] give an algorithm for computing a perfect token distribution assuming that each node at the beginning of the computation knows the number of nodes in the tree.

Our results. We present a fully decentralized algorithm for tree-connected networks which computes a perfect token distribution in time $O(TD)$ without receiving as input any additional (aggregate) information about the network such as the total number of nodes or the total number of tokens. The number of token exchanges made by our algorithm exceeds the optimum by at most $O(D \min\{T, N\})$. In addition, at the end of the redistribution process, all the nodes reach a distinguished final state and they are ready to start a new activity.

Our technique is based on a three phase approach.
- *Phase 1.* We compute (in a completely distributed way) the number of nodes of the tree and the number of tokens scattered across the tree in $\Theta(D\,d)$, where d is the maximum degree and D is the diameter of the tree[4].
- *Phase 2.* Let v be any node of the tree and T_i, $1 \leq i \leq k$, be the k sub-trees rooted at nodes adjacent to v. After Phase 1 v knows the number of nodes and the number of tokens contained in each T_i. Taking advantage of this information each node having more than $\lfloor T/N \rfloor$ tokens is able to decide where to send its

[4] in multi-port models such information can be computed in $\Theta(D)$.

extra tokens. Nodes that have less than $\lfloor T/N \rfloor$ tokens wait until they receive enough tokens from their neighbors. After Phase 2 we get an "almost perfect" token distribution in which at least $N-1$ nodes have exactly $\lfloor T/N \rfloor$ tokens. Note that almost perfect token distributions have discrepancy at most (T mod N).
- *Phase 3.* We refine the distribution obtained after Phase 2 in order to make it perfect. To this extent, the node containing more than $\lceil T/N \rceil$ tokens (at most one node has this property) sends them to its neighbors. This procedure is then executed recursively by all the neighbors of v.

The rest of the paper is organized as follows. In Sect. 2 we define some communication primitives that will be used in our algorithm. In Sect. 3 we present our algorithm while in Sect. 4 and 5 we sketch the proof of its correctness and we discuss its computational cost in terms of time, space, and number of token exchanges, respectively. Section 6 contains conclusions and a brief description of possible open problems.

2 Communication Primitives.

We define the following communication primitives:
-*reserve*(c) reserves channel c for communication and returns *busy* if c is already reserved;
-*release*(c) releases channel c reserved for communication;
-*send*(c,m) sends message m through the already reserved channel c;
-*receive*(c) receives a message from channel c if a message was sent on channel c (channel needs not to be reserved). If no message is arriving from c the message returned by *receive* is *NULL*. If two or more messages are sent on a channel then they are received one by one in the order they were sent.

Using these primitives we build functions:
-*waitany*(c) interrogates every channel until a message is received, then returns the message received, saving the channel through which it arrived in the parameter c;
-*wait*(c) interrogates channel c until a message is received, then returns the message received; -*safeSend*(c,m) waits until c is not *busy*, reserves it, sends m through c, and then releases c;
-*receiveToken*(c) receives a token from c, if any. If a message is received updates local variables containing the number of tokens;
-*receiveTokensFromAll*() receives, using *receiveToken*(c), tokens sent from neighbors, if any;
-*sendToken*(c) sends, using *safeSend*(c,m) a token from the current node through c and updates the local variables containing the number of tokens.

3 The Algorithm.

Our algorithm consists of three phases: first it collects information about the tree structure and the initial token distribution, then uses this information for

obtaining an almost perfect token distribution, and finally it refines the solution by redistributing a small number of tokens.
In what follows we assume that each node v can store $\Theta(d_v(\log T + \log N))$ bits of information.

3.1 Phase 1.

Each node v has an internal state defined according to the values of two local variables, namely S and R which contain the number of messages sent and received by v so far. Let deg be the degree d_v of v. Both S and R are set to 0 at the beginning of the computation. According to the values of S and R we define 5 distinct states:
$S1.1$: **[**$\mathsf{S} = 0$ and $\mathsf{R} < (\mathsf{deg} -1)$**]**. All the nodes having degree greater than 1 start in this state. Nodes in this state are waiting for messages from their neighbors. Any node with k neighbors waits for $k-1$ messages and then changes state (from $S1.1$ to $S1.2$).
$S1.2$: **[**$\mathsf{S} = 0$ and $\mathsf{R} = (\mathsf{deg} -1)$**]**. Leaves start in this state. Each node with k neighbors reaches this state after receiving $k-1$ messages. A node in this state tries to send information about its subtree to the only neighbor from which it did not receive any message yet. If succeding the node changes state (from $S1.2$ to $S1.3$). Otherwise, if the channel is busy then the last neighbor is sending subtree information. The node receives it and changes state (from $S1.2$ to $S1.4$).
$S1.3$: **[**$\mathsf{S} = 1$ and $\mathsf{R} = (\mathsf{deg} -1)$**]**. Nodes in this state are waiting for global information from the neighbor to which they sent their local information.
$S1.4$: **[**$\mathsf{S} = 0$ and $\mathsf{R} = \mathsf{deg}$**]**. Only one node reaches this state: the one computing global information. In this state it sends such information to all its neighbors and then ends.
$S1.5$: **[**$\mathsf{S} = 1$ and $\mathsf{R} = \mathsf{deg}$**]**. In this state a node has already received global information and forwards it to all its neighbors and then ends.

In phase 1 each message msg consists of two integer numbers, referred to as $\mathsf{msg.T}$ and $\mathsf{msg.N}$ representing the number of tokens and of nodes of the entire subtree rooted at the sender of msg. We assume that channels at each node v are numbered from 1 to deg.

3.2 Phase 2.

In phase 2 we start moving tokens.
From now on we will refer to the node which received local information from all its neighbors as the root of the tree. Every node has two arrays, namely $\mathsf{subTreeN}$ and $\mathsf{subTreeT}$, containing the total number of nodes and the total number of tokens in the subtrees rooted at its neighbors. Each node, knowing total number of tokens T and of nodes N in the tree, computes the average number of tokens per node $\mathsf{Avg} = \lfloor T/N \rfloor$. All nodes send or wait for tokens until they and the nodes in their subtrees contain exactly Avg tokens, except for the

Phase 1

initialization

```
deg ← d_v                                    //neighbors of v
T_v ← tokens at v                            //tokens in subtree rooted at v
N_v ← 1                                      //nodes in subtree rooted at v
R ← 0                                        //messages received by v
S ← 0                                        //messages sent by v
parent ← 0                                   //the parent of the node
subTreeT array of deg NULL elements          //tokens in neighbor subtrees
subTreeN array of deg NULL elements          //nodes in neighbor subtrees

while(S < deg or R < deg) do
  case (S,R) of
  S = 0, R < (deg − 1):
    while(R < (deg − 1)) do      //receive local info from all neighbors but one
     | msg←waitany(i)            //waits for a message from any neighbor
     | T_v ←T_v+msg.T            //tokens in subtree rooted at v
     | N_v ←N_v+msg.N            //nodes in subtree rooted at v
     | subTreeT[i]←msg.T         //tokens in subtree rooted at v_i
     | subTreeN[i]←msg.N         //nodes in subtree rooted at v_i
     | R←R+1                     //v received a message
    endwhile
  S = 0, R = (deg − 1):
   parent ← channel j for which subTreeT[j]=NULL     //the channel from which v did not received msg
    if (reserve(parent) ≠busy) then   //check that v_i is not using the channel
     | msg←receive(parent)            //check if received the message
     | if (msg=NULL) then             //message not received
     |  | msg.T←T_v                   //nodes in subtree rooted at v
     |  | msg.N←N_v                   //tokens in subtree rooted at v
     |  | send(parent,msg)            //send values of subtree rooted at v to v_parent
     |  | release(parent)             //release channel parent
     |  | S←S+1                       //v sent a message
     | else
     |  | release(parent)             //release channel parent
     |  | T_v ←T_v+msg.T              //tokens in subtree rooted at v
     |  | N_v ←N_v+msg.N              //nodes in subtree rooted at v
     |  | subTreeT[parent]←msg.T      //tokens in subtree rooted at v_parent
     |  | subTreeN[parent]←msg.N      //nodes in subtree rooted at v_parent
     |  | parent← 0                   //v is root, parent = 0
     |  | R←R+1                       //v received a message
     | endif
    else
     | msg←wait(parent)               //v_parent using the channel : wait for the message
     | T_v ←T_v+msg.T                 //tokens in subtree rooted at v
     | N_v ←N_v+msg.N                 //nodes in subtree rooted at v
     | subTreeT[parent]←msg.T         //tokens in subtree rooted at v_parent
     | subTreeN[parent]←msg.N         //nodes in subtree rooted at v_parent
     | parent← 0                      //v is root, parent = 0
     | R←R+1                          //v received a message
    endif
  S = 1, R = (deg − 1):
   msg←wait(parent)                   //waits for global message
   subTreeT[parent]←msg.T−T_v         //tokens in subtree rooted at v_parent
   subTreeN[parent]←msg.N−N_v         //nodes in subtree rooted at v_parent
   T_v ←msg.T                         //total tokens of the tree
   N_v ←msg.N                         //total nodes of the tree
   R←R+1                              //v received a message
  S = 0, R = deg:
   msg.T←T_v                          //total tokens of the tree
   msg.N←N_v                          //total nodes of the tree
    for i← 1 to deg do                //for all neighbors v_i
     | safeSend(i,msg)                //send global values to v_i
    endfor
   S←deg                              //v sent deg messages
  S = 1, R = deg:
   msg.T←T_v                          //total tokens of the tree
   msg.N←N_v                          //total nodes of the tree
    for i← 1 to deg do                //has to send global values to all neighbors
     | if (i ≠ parent) then           //except v_parent
     |  | safeSend(i,msg)             //send global values to v_i
     | endif
    endfor
   S←deg                              //v sent deg − 1 messages
  endcase
endwhile
```

root of the tree which might contain extra tokens. Function $subTreeAvg(\mathsf{i})$ is a simple function computing the average number of tokens in the subtree rooted at v_i: it returns subTreeT[i]/subTreeN[i]. Each node v in Phase 2 has an internal state represented by
- a boolean variable subTreeBalanced ($true$ if all the subtrees rooted at children of v are balanced, $false$ otherwise);
- an integer variable T_v storing the number of tokens at v;
- an integer variable parent which is set to 0 if v is the root.

$S2.1$: **[subTreeBalanced** $= false$ and $\mathbf{T}_v \leq$ **Avg]**. Nodes remain in this state receiving tokens and updating local variables until they have more than Avg tokens and then change state. (from $S2.1$ to $S2.3$).
$S2.2$: **[subTreeBalanced** $= true$ and $\mathbf{T}_v <$ **Avg**, **parent** > 0**]**. Nodes in this state need not to exchange tokens with children, since all such subtrees have Avg number of tokens. They wait for tokens from their parent (a node in this state is not the root).
$S2.3$: **[subTreeBalanced** $= false$ and $\mathbf{T}_v >$ **Avg]**. Nodes in this state need to balance their descending subtrees: they receive tokens from any node and then send tokens to subtrees with less than Avg number of tokens.
$S2.4$: **[subTreeBalanced** $= true$ and $\mathbf{T}_v >$ **Avg**, **parent** > 0**]**. Nodes in this state have already balanced subtrees rooted at their children and send extra tokens towards their parents.
As we already mentioned, a node stops exchanging tokens if it has balanced its subtrees (subTreeBalanced $= true$ and $\mathsf{T}_v =$ Avg) (Fig. 1 part (b)) or if it has balanced the subtrees of all its children (subTreeBalanced$= true$) and it is the root (parent $= 0$)) (Fig. 1 part (c)).

3.3 Phase 3.

In this phase tokens keep on moving until a *Finished* message is sent.
In order to distinguish between tokens and finished messages we use two internal variables, namely msg.Token and msg.Finished. At any time if msg.Finished $= true$ then msg.Token $= NULL$, else if msg.Finished $= false$ then msg.Token contains a token.

In phase 3 extra tokens sent to the root during phase 2 are redistributed down the tree. The root (and all the other nodes recursively) sends its extra tokens to subtrees which can accept them.[5] In this phase each node can be in one of the two following states:
$S3.1$: **[$\mathbf{T}_v >$ Avg**$+1$ and **parent** $= 0$**]**. Nodes in this state are root of a subtree and send their extra tokens to subtrees which can accept them.
$S3.2$: **[$\mathbf{T}_v$ any value** and **parent** > 0**]**. Nodes in this state receive messages from their parent. If they receive tokens they update local variables. If they receive a *Finished* message they set parent to 0 and become a root.
When a node is root and has at most Avg$+1$ number of tokens it sends a *Finished*

[5] eventually, each node ends up with either Avg or Avg$+1$ tokens, then each subtree must contain less than Avg$+1$ tokens.

```
PHASE 2
INITIALIZATION
Avg ← ⌊ T_v/N_v ⌋
T_v ← tokens at v
R ← 0
bigger ← {i|subTreeAvg(i) > Avg, i ≠ parent}
smaller ← {i|subTreeAvg(i) < Avg, i ≠ parent}
subTreeBalanced← (bigger= ∅ and smaller= ∅)

while(not((subTreeBalanced=true and T_v=Avg)
      or (subTreeBalanced=true and parent= 0)) do
  case (subTreeBalanced, T_v , parent) of
   subTreeBalanced = false, T_v ≤ Avg, parent any value
      receiveTokensFromAll()          //check for incoming tokens and update data
      bigger ← {i|subTreeAvg(i) > Avg, i ≠ parent}
      subTreeBalanced ← (bigger= ∅ and smaller= ∅)
   subTreeBalanced = true, T_v < Avg, parent > 0
      receiveToken(parent)            //all sons have finished,
                                      //checking tokens coming from parent
   subTreeBalanced = false, T_v > Avg, parent any value
      receiveTokensFromAll()          //check for incoming tokens and update data
      bigger ← {i|subTreeAvg(i) > Avg, i ≠ parent}
      if(smaller≠ ∅)
        j ← an element ∈ smaller
        sendToken(j)                  //send token to v_j and update data
        smaller ← {i|subTreeAvg(i) < Avg, i ≠ parent}
      endif
      subTreeBalanced ← (bigger= ∅ and smaller= ∅)
   subTreeBalanced = true, T_v > Avg, parent > 0
      sendToken(parent)               //send token to v_parent and update data
  endcase
endwhile
```

message to all its children. As proven in Sect. 4, eventually all nodes become a root and receive a *Finished* message (Fig. 1 part (d)).

4 Proof of Correctness.

In this Section we prove the correctness of our algorithm.

4.1 Phase 1.

Let $S1.final$ be the final state of Phase 1 defined by [S =deg and R =deg].

Theorem 1. *At the end of phase 1, $\forall v \in V$ we have that $\mathsf{T}_v = T$ and $\mathsf{N}_v = N$. In addition, exactly one node (the root of the tree) has* parent $= 0$ *while each other node has* parent $= i$ *where i is the number of the channel which links the node to its parent in the rooted tree.*

Sketch of the proof. At the beginning of the computation, each node of degree 1 is in state $S1.2$ while all the other nodes are in state $S1.1$. Then, each node in state $S1.2$ sends its subtree information to nodes in state $S1.1$ and moves to state $S1.3$. Eventually each node in state $S1.1$ moves to state $S1.2$ and decides which node is its parent. When state $S1.2$ contains only two nodes, one of them moves to state $S1.4$. All the other nodes go to state $S1.3$ and wait for global information. The node in state $S1.4$ is the root, computes global information

```
PHASE 3
INITIALIZATION
p ← parent                                  //p is the parent of v

while(not(T_v ≤Avg+1 and parent= 0)) do
 case (T_v , parent) of
  T_v > Avg+1, parent = 0
     i ← a neighbor for which subTreeAvg(j) < Avg + 1,  i ≠p
     msg.Token ← a token at v
     msg.Finished ← false
     safeSend(i,msg)                        //send token to v_i
     T_v  ← T_v  − 1                        //tokens at v
     subTreeT[i] ← subTreeT[i]+1            //tokens in subtree rooted at v_i
  T_v any value, parent > 0
     msg ← receive(parent)
     if(msg≠NULL)                           //if a message msg received
       if(msg.Finished=true)                //if message received was Finished
        | parent ← 0                        //the parent finished : become root
       else                                 //else a token was received
        | T_v  ← T_v  +1
        | subTreeT[parent] ← subTreeT[parent]−1
      endif
     endif
 endcase
endwhile
for i← 1 to deg do                          //for all neighbors v_i
   | msg.Token ← NULL
   | msg.Finished ← true
   |  if(i≠p)                               //except the parent
   |    | safeSend(i,msg)                   //send Finished to v_i
   |  endif
endfor
```

and sends it to its neighbors. Neighbors receive such information, identify the node which sent them as they parent, move to state $S1.5$, and then forward it to their children. At the end of this process all the nodes receive global information and set $\mathsf{T}_v = T$ and $\mathsf{N}_v = N$. Eventually each node moves to state $S1.final$. $\square$

4.2 Phase 2.

During Phase 1 we implicitly elect a leader, namely the only node with parent $= 0$. We refer to this node as the root of the tree. Links to parent nodes are provided by local variables parent.

Lemma 1. *Let v_j be a node different from the root of the tree and such that $subTreeAvg(j)$*[6] *$\geq$ Avg. Let v be any node of the subtree rooted at v_j. Then at the end of phase 2 v contains exactly Avg tokens.*

Proof. We make the proof by induction on the height of the subtree:
Base case. If the subtree has height 0 (only one node) subTreeBalanced $= true$. If the node has Avg tokens it will do nothing in Phase 2 and the lemma is true, if it has more it will send tokens to its parent (state $S2.4$) until it remains Avg tokens.
Induction. If the subtree has height $n + 1$ it has subsubtrees of height

[6] each neighbor of v_j has a different value for $subTreeAvg(j)$ but, being the tree rooted, we refer the value owned by the neighbor v_{parent} which is the parent of v_j.

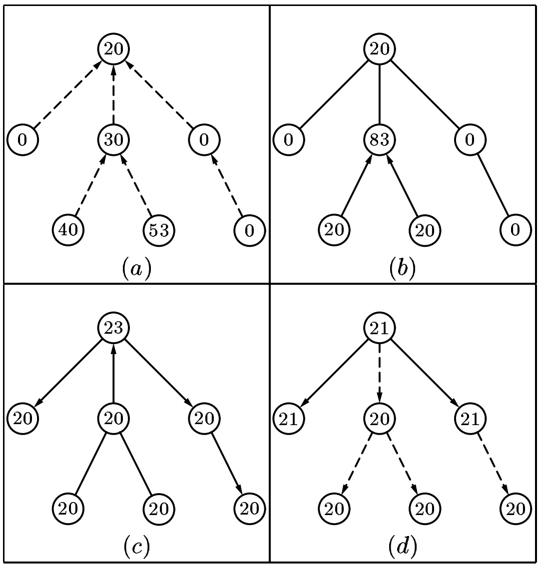

Fig. 1. Example of execution of our algorithm. Dashed arrows represent information messages, solid arrows represent token exchanges. (a) Phase 1: local information is sent from leaves to a single node which becomes the root of the tree. (b) Phase 2: token are redistributed in subtrees until each subtree has Avg number of tokens. (c) End of Phase 2: each node has Avg tokens but the root that may contain some extra token. (d) End of Phase 3: a perfect token distribution is achieved and each node receives a *Finished* message.

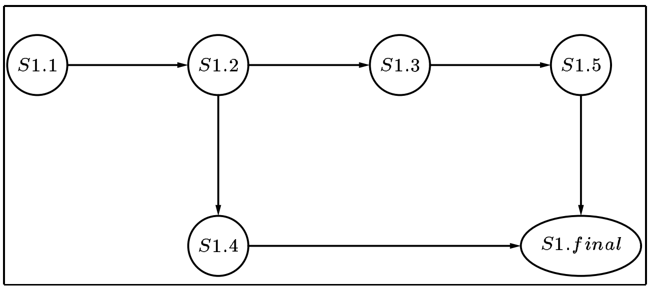

Fig. 2. State transitions diagram of phase 1.

n. Since the algorithm behaves exactly in the same way for every node in the tree except the root we can apply induction hypothesis on them. When subTreeBalanced $= true$ the root of the subtree has at least Avg tokens and sends extra tokens toward the root of the tree as in the base case (state $S2.4$) and every other node finishes with Avg tokens for induction hypothesis, so the lemma holds. If subTreeBalanced $= false$ for induction hypothesis all subsubtrees with $subTreeAvg \geq$ Avg will have Avg tokens in each node and send extra tokens to the root of the subtree. The root will receive such tokens (states $S2.1$ and $S2.3$) and, since the total number of tokens of the subtree is at least subTreeN[j] $\cdot$ Avg, will eventually have at least Avg tokens. If there are subsubTrees with $subTreeAvg <$ Avg their root can not be in state $S2.4$ and so will receive tokens from their parent. Their parent, the root of the subTree, has more than Avg tokens and sends some of them to such subsubtrees (state $S2.3$). Reasoning again on number of tokens the root will make such subsubtrees to have $subTreeAvg =$ Avg remaining at least Avg tokens. At this point subTreeBalanced will become $true$ and the lemma holds, as explained above. □

Theorem 2. *At the end of phase 2 the root of the tree has at least Avg tokens while all the other nodes have exactly Avg tokens.*

Proof. Consider subTreeBalanced $= true$ at the root of the tree. Then we apply lemma 1 to all the children v_j of the root: all nodes in their subtrees have exactly Avg tokens. Then, for definition of Avg, the root has at least Avg tokens and the theorem holds. Consider subTreeBalanced $= false$ at the root of the tree. Then we apply lemma 1 to all the children v_j of the root having $subTreeAvg(\mathrm{j}) \geq$ Avg. During phase 2 the root receives tokens (states $S2.1$ and $S2.3$) from subtrees having $subTreeAvg(\mathrm{j}) >$ Avg. Since the total number of tokens of the tree is at least $N\cdot$ Avg, when such subtrees reach balancing the root has at least Avg tokens. If there are subTrees having $subTreeAvg(\mathrm{j}) <$ Avg the root will have more than Avg tokens and will send some of them to such subsubtrees (state $S2.3$). The root v_j of such subtrees can not be in state $S2.4$ and then receives tokens sent from the root. Reasoning again on number of tokens the root will make such subsubtrees to have $subTreeAvg(\mathrm{j}) =$ Avg remaining at least Avg tokens. At this point subTreeBalanced will become $true$. We apply lemma 1 to all the children v_j of the root and the theorem holds. □

4.3 Phase 3.

Theorem 3. *After phase 3 each node in the tree has Avg or Avg* $+1$ *tokens.*

Proof. We make the proof by induction on the height of the tree:
Base case. If the tree has height 0 (only one node) being the total number of tokens at most Avg $+1$ the theorem is trivially true.
Induction. If the subtree has height $n + 1$ it has subsubtrees of height n. If the root has at most Avg $+1$ tokens the theorem is proved for theorem 2. If it has more tokens it sends them to its sons until it remains Avg $+1$ tokens, due to the total number of tokens this will happen before that all subtrees have $subTreeAvg =$ Avg $+1$. At this time the root will send $Finished$ to all its sons. On the other hand each son receives tokens without sending them and sets parent to 0 upon receiving the $Finished$ message. Now the algorithm repeats recursively on each subtree rooted at each son of the root. Since the average number of tokens for each subtree is at most Avg $+1$ the total number of tokens in the subtree is such that its root can send all extra tokens to sons before their

subtrees have more than $\mathsf{Avg} + 1$ average number of tokens. We can then apply induction hypothesis to every subtree proving the theorem. □

5 Complexity.

We call d the maximum degree of the tree, T the total number of tokens, D the diameter of the tree and N the number of nodes. Phase 1 runs in (worst case) $\Theta(Dd)$ in single-port model[7]. Phase 2 runs in (worst case) $O(TD)$, while phase 3 runs in (worst case) $O(D(T \bmod N))$. Since phase 2 has the greatest complexity the overall complexity of the algorithm is $O(TD)$.
The space needed to store local information 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 $\Theta(d(\log T + \log N))$.

In phase 2 all tokens travelling through a link go in the same direction and, if $\mathsf{Avg} = T/N$, algorithm achieves perfect token distribution. This means that the number of token exchanges is optimum. If $\mathsf{Avg} < T/N$ then at most $T \bmod N$ tokens generate extra token exchanges. The longest path done by such tokens is from a leaf to the root. So the number of token exchanges is at most $O(D(T \bmod N))$ greater than optimum. In phase 3, unless extra tokens were already at the root, we have already done more token exchanges than the optimum, so all the $O(D(T \bmod N))$ worst case token exchanges are beyond the optimum. The total number of token exchanges is then no more than $O(D \min\{T, N\})$ greater than optimum.

6 Conclusions and Further Work.

We presented a distributed algorithm which computes a perfect token distribution on anonymous tree connected networks without taking as input any global information about the tree structure or about the initial distribution of tokens.

Some other questions remain open and deserve further investigation. Among them we wish to point out the following two that are strictly related to the results presented in this paper.
(1) Each node in our algorithm has a local memory of size $\Theta(d(\log T + \log N))$. Is it possible to compute perfect token distributions on trees assuming that each node has only constant size local storing capacity?
(2) Is it possible to compute perfect token distribution in a fully decentralized way minimizing the number of token exchanges?

References

1. J. E. Boillat. Load balancing and Poisson equation in a graph. *Concurrency: Practice and Experience*, 2(4):289–311, 1990.

[7] $\Theta(D)$ in multi-port model.

2. G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.
3. Johannes E. Gehrke, C. Greg Plaxton, and Rajmohan Rajaraman. Rapid convergence of a local load balancing algorithm for asynchronous rings. *Theoretical Computer Science*, 220:247–265, 1999.
4. B. Ghosh, F. T. Leighton, B. M. Maggs, S. Muthukrishnan, C. G. Plaxton, R. Rajaraman, A. W. Richa, R. E. Tarjan, and D. Zuckerman. Tight analyses of two local load balancing algorithms. In *Proc. of the 27th Annual ACM Symposium on Theory of computing*, 548–558, May 1995.
5. M. E. Houle, A. Symvonis, and D. R. Wood. Dimension-Exchange Algorithms for Token Distribution on Tree-Connected Architectures. *Journal of Parallel and Distributed Computing*, to appear. Also in *Proc. of 9th International Colloquium on Structural Information and Communication Complexity (SIROCCO '02)*, 181–196, Carleton Scientific, 2002.
6. M. E. Houle, E. Tempero, and G. Turner. Optimal dimension-exchange token distribution on complete binary trees. *Theoretical Computer Science*, 220(2):363–376, 1999.
7. M. E. Houle and G. Turner. Dimension-exchange token distribution on the mesh and the torus. *Parallel Computing*, 24(2):247–265, 1998.
8. B. Shirazi, A. Hurson, and K. Kavi. Scheduling and local balancing in parallel and distributed systems. *IEEE computer society press*, 1995.
9. J. Song. A partially asynchronous and iterative algorithm for distributed load balancing. *Parallel Computing*, 20:853–868, 1994.
10. R. Subramanian and I. D. Scherson. An analysis of diffusive load-balancing. *ACM Symposium on Parallel Architectures and Algorithms*, 220–225, 1994.
11. A. N. Tantawi and D. Towsley. Optimal static load balancing in distributed computer systems. *Journal of the ACM*, 32:445–465, 1985.