Distributed incremental maximum finding in hierarchically divided graphs

P.J.A. Lentfert, S.D. Swierstra, A.H. Uittenbogaard

RUU-CS-90-30
September 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Distributed incremental maximum finding in hierarchically divided graphs

P.J.A. Lentfert, S.D. Swierstra, A.H. Uittenbogaard

RUU-CS-90-30
September 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

ISSN:0924-3275

Distributed Incremental Maximum Finding in Hierarchically Divided Graphs

P.J.A. Lentfert, S.D. Swierstra
Department of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht,
The Netherlands.

A.H. Uittenbogaard
HCS Industrial Automation B.V.
P.O. Box 20020, 7302 HA Apeldoorn,
The Netherlands.

1 Introduction

Distributed algorithms are often informally described. An advantage of an informal description is the seeming ease with which they can be produced and studied. It is not necessary to understand a formal system, before finding out the ideas behind the algorithm.

However, a major disadvantage in providing only informal arguments is that errors are easily made. It is our experience that distributed algorithms which seemed to be correct, and were even informally proven to be correct, in fact were wrong.

For example, during our research ([3]), we once were convinced to have designed a correct distributed algorithm. By coincidence, we noticed that in some cases the update algorithm did not terminate. In our proof we showed that the algorithm would terminate after any update, by presenting a function whose value decreased in every step of the algorithm and was bounded from below. Unfortunately, this function could have an infinite value, in which case the lower bound would never be reached.

It was decided to use a formal method to be more sure of a correct outcome. UNITY ([1]) was chosen, a formalism which is claimed to be helpful in the design of distributed and concurrent programs. In [1], Hoare claims that UNITY is a complete theory of programming because it includes the following important aspects:

- It is a method for specification of programs.
- It is a method for reasoning about specifications.
- It is a method of developing programs together with a proof that they meet their specification.
- It is a method of transforming programs to achieve high efficiency on the machines available for their execution.

The design and study of algorithms using a formalism can be extremely tiresome, although rewarding. It is our experience that formal methods (including UNITY) soon become very complex when studying nontrivial problems. Therefore, an informal description may be very helpful.

In [3] we introduced the distributed hierarchical routing algorithm that operates on a hierarchically divided network, i.e., a network that is divided in domains and each domain, in turn, is divided in subdomains, etc. As a result of hierarchically dividing the network, the size of the routing tables remains small. The size of the tables for nodes in a hierarchically divided network containing N nodes can be reduced to $O(\log N)$ entries.

It is our goal to formally design a distributed update algorithm on a hierarchically divided network upon which the distributed hierarchical routing algorithm operates. Before using UNITY on our update algorithm, we have decided to first gain some experience in using UNITY on a simpler problem. This algorithm to be derived is used to obtain and maintain the maximum value of a set of frequently changing inputs. Every node in a hierarchically divided network is associated with precisely one element of this set of inputs, and every node has to maintain the maximum value of the inputs of all nodes.

This problem and the problem of keeping topological information about a hierarchically divided network bear some similarities. The network can be considered as a set of nodes. With each node in this set corresponds a possibly changing set of paths from a source node to that node. Every node must, for every other destination node in the set, obtain and maintain the best path of the set of paths to that destination node, according to some metric (for example the hop-count).

The problem of maintaining this routing information is for two reasons more complex than the problem of keeping the maximum value. In the first place, it deals with a set of sets, viz., for each node in the set of destination nodes there is a set of paths to it.

The second reason is that every node has to deal with a different set of paths for the same destination node. In keeping the maximum value of a set of inputs, each node has to deal with one, and the same, set of integers.

We are, however, convinced that the (formal) derivation of a distributed algorithm for keeping the maximum value of a frequently changing set of inputs, that operates on a hierarchically divided network, will help us in the (formal) derivation of an update algorithm.

In Section 2 the topological model is described. The description of the algorithm consists of two parts. Because, as stated above, an informal description may be very helpful, we have included in Section 3, a traditional description and solution of the problem. In that part we develop an algorithm by first designing an abstract algorithm that operates on a virtual datastructure. In the next step, this algorithm is transformed into a distributed algorithm that operates on a hierarchically divided network. Also a more traditional proof of the distributed algorithm is given.

In Section 4, these steps are formalized.

2 Model

In this article we describe a distributed algorithm on a network that is identified with a bidirectional graph, the *network graph*. The nodes of this graph represent processors in the network, the edges represent communication links between them. Every node has local computing capacity and memory to which no other node has (direct) access. Nodes cooperate by exchanging messages over the links connecting them. These links are assumed to be fault-free and have a FIFO behaviour.

Upon the graph a hierarchical structure is imposed ([3]): nodes are grouped into domains, which, in turn, are grouped into superdomains, etc. Nodes are viewed as the smallest possible domains. The graph, thus divided, is called a *hierarchically divided graph*. The division of the graph is modeled as a tree, called the *division tree*. The network domain (the domain comprising the whole network) corresponds to the root of this tree. Subdomains of a domain correspond to sons of the tree-node corresponding to the domain. The leaves of the division tree correspond to the processors in the network.

When tree-node m in the division tree is a son of tree-node n, we write: $m \lhd n$. Analogously, when domain C in the hierarchically divided network is a subdomain of domain D, we write: $C \lhd D$. Tree-node n in the division tree is *visible* to tree-node m, if n is the son of a tree-node on the path in the division tree to m, but is not on that path itself. Analogously, domain D in the hierarchically divided network is visible to domain C if D is a direct subdomain of a domain E of which C is a descendant, too, and C is not contained in D.

Note that, by definition, if a domain D is visible to domain C, then D is visible to every node in C. Also, subdomains of a domain that is visible to a node are not visible to that node. As a consequence, internal details of a domain are hidden from domains, especially from nodes, outside that domain. This will result in a reduction of the storage complexity of the algorithms to be

developed: only a single item of information has to be stored at a node n for each domain visible to n, and not for every single node in it.

On the hierarchical division we impose the following restriction: every domain is connected. In other words: a message routed between two nodes does not have to leave the least common domain of those nodes. In order to meet this constraint, it is possible, as described in [3], that subdomains are shared between domains, i.e., domains may be overlapping.

3 Traditional Description

3.1 Problem

In this section we describe the problem that is solved in the rest of this article.

We assume that every node in a hierarchically divided network has an associated data item, called its *input*. The inputs are, not necessarily unique, elements from a totally ordered set. Furthermore, every node possesses a variable in which it keeps the maximum of all inputs in the network.

Inputs of nodes may change spontaneously, and as a result the global maximum may change. If this is the case, all nodes in the network are required to have updated their maximum value within finite time after the update has occurred. If multiple changes occur, nodes must have updated their maximum value within finite time after the last update.

3.2 Update Algorithm on a Division Tree

In this section we present an update algorithm that operates on a division tree after the input of a leaf has changed. In this algorithm, leaves as well as internal tree-nodes of the division tree are assumed to be capable of performing actions and of storing values (of variables). Keep in mind, however, that in a physical network, nodes are the only entities having these properties. Since internal tree-nodes in the division tree model domains in the hierarchical division of the network, the actions of the domains have to be emulated by the physical nodes "on behalf of the domains."

The following questions, then, have to be answered. Which nodes simulate the actions corresponding to a domain? Do we need special nodes, or can the domain actions be established by all nodes (in the domain) constituting a part in it?

In Section 3.3 we will describe a transformation of the abstract algorithm, to an algorithm in which the only active entities are nodes in a hierarchically divided network.

Every tree-node n in the division tree maintains the following variables:

$f(n)$: the father of tree-node n in the division tree,

$m(n)$: the *m-value* of n, defined as the input of n if n is a leaf, and (an approximation of) $\max\{m(s) \mid s \lhd n\}$ if n is an internal tree-node, and

$m_n(s)$: the current estimate of $m(s)$, for every son s of n, if n is an internal tree-node.

We assume a correctly initialized datastructure, i.e., every leaf n has got an input $m(n)$, and for every internal tree-node n, $m(n) = \max\{m_n(s) \mid s \lhd n\}$ and for every son s of n, $m_n(s) = m(s)$. Thus, $m(r)$ contains the maximum value of all leaves in the tree if r is the root of the division tree.

We now describe the distributed algorithm that is performed as a result of a change in the input of a leaf. The update algorithm is straightforward. Upon detecting a change in $m(n)$, tree-node n reports this change to $f(n)$. In response, $f(n)$ updates its value and, if it changes, recursively, reports this to $f(f(n))$.

The resulting algorithm is presented slightly more formally below.

Algorithm I (division tree)

- Upon detecting a change in $m(n)$ to m', leaf n sends message $UPD(n, m')$ to $f(n)$.
- Upon receipt of $UPD(s, m')$, internal tree-node n performs the following actions:

  $m_n(s) := m'$;
  $m(n) := \max\{m_n(s_i) \mid s_i \lhd n\}$;
  if $m(n)$ changed and n is not the root
  then send $UPD(n, m(n))$ to $f(n)$
  fi

□

3.3 Update Algorithm on a Hierarchically Divided Graph

In this section we will distribute the algorithm on a division tree as described in Section 3.2. A domain (an internal tree-node in the division tree) is simulated by the cooperation of all nodes contained in it. For that purpose we distribute the information kept by any internal tree-node D in the division tree in Algorithm I over the nodes contained in domain D in the hierarchically divided network. This is realized by a "projection" in the division tree of all information onto every leaf in the subtree below the tree-node containing the information. As a result, every node keeps the m-values of its superdomains *and* the m-values of its visible domains.

We denote by $m_n(D)$ the m-value of domain D stored at node n. In a stable state (for example before any changes occur) all nodes in domain D will have the same value for $m_n(D)$.

In Section 3.3.1 we handle the update of data after a single change has occurred. In Section 3.3.2 we extend the algorithm to perform the necessary actions after multiple changes have occurred "at the same time." In Section 3.3.3 this algorithm is proven to be correct.

3.3.1 Updating a Single Change

In the following description, let n be a node with direct superdomain D, which has direct superdomain F (i.e., $n \lhd D \lhd F$). If n's input changes to m'_n, every brother of n, i.e., every node d in D, has to update its value $m_d(n)$. This is established by broadcasting within D the message $UPD(n, m'_n)$, using Propagation of Information, see [5].

The change of n's input may affect the m-value of D. For example, if n's input was the maximal input in D, and it decreased, D's m-value decreases too (unless another node in D has this same maximal input). An increase of n's input on the other hand, may result in an increase in D's m-value. Anyway, if D's m-value changes, the new m-value can be determined by every node d inside D by computing $m_d(D) = \max\{m_d(n_i) \mid n_i \lhd D\}$. This changed value has to be reported to all other nodes in F. This is established by a broadcast of $UPD(D, m'_D)$, where m'_D is the new m-value of D. Nodes in D with neighbour-nodes outside D but in F, called *border nodes* of D, start broadcasting the new m-value.

Analogously, this change in D may affect F's m-value, necessitating the same actions of the nodes in F and in the direct superdomain of F. This procedure is repeated on every level for which the change has an effect.

The resulting algorithm is presented below. The notation $\{n, n'\}$ is used to denote a link between nodes n and n'.

Algorithm II (network, single change)

- Upon detecting a change in $m(n)$ to m'_n, node n performs $\text{Accept}_n(n, m'_n)$.
- Upon receipt of $UPD(D, m_D)$, node n performs the following actions:

  if $m_n(D) \neq m_D$
  then $\text{Accept}_n(D, m_D)$
  fi,

where

$\text{Accept}_n(D, m_D) =$
Let $D \lhd F$ **in**
$\{m_n(D) := m_D;$
for all $l = \{n, f\}, f \notin D \wedge f \in F$
do send $UPD(D, m_D)$ **to** f
od;
if $m_n(F) \neq \max\{m_n(D_i) \mid D_i \lhd F\}$
then /* new round necessary */
$\text{Accept}_n(F, \max\{m_n(D_i) \mid D_i \lhd F\})$
fi
$\}$

□

Note, that the algorithm requires at most two update messages to be sent over every link in the network. Moreover, the update messages are confined to the lowest level domain with a subdomain of which the input changes.

3.3.2 Updating After Multiple Changes

In this section we extend Algorithm II to deal with the case of multiple changes of inputs occurring at the same time. When the input of a node changes, the node starts reporting this change using Algorithm II. It is possible, however, that during this process another node changes its input, too. Since the information about this change will not immediately reach the first node, this node may start a broadcast based on out-of-date information. Therefore, update messages reporting different m-values of the same domain may be in transit at the same time.

As before, in order to simulate the actions of a domain by the cooperation of the nodes in it, every node in the domain maintains the m-value of each of its subdomains. Because of this projection of all information in the division tree onto every leaf, nodes are not aware of the internal structure of their visible domains. Specifically, nodes do not keep the m-values of subdomains of their visible domains. Nodes, therefore, are *not* able to determine whether the information received from a visible domain is up-to-date or not (i.e., whether the value equals the maximum of the inputs of the subdomains).

To distinguish between succeeding broadcasts for a domain we introduce a sequence number ([5]) per domain. In [4], Perlman describes a broadcast algorithm of routing information that stabilizes in reasonable time without human intervention after any malfunctioning equipment is repaired or disconnected. It is shown why in a network without globally synchronized clocks the sequence number scheme is better suited than local clocks for determining whether a received packet is older or newer than the last received packet. The problems with sequence numbers, such as the possibility of wrap-around, must also be considered using logical clocks. However, the logical clock scheme requires hardware that is not always available in packet switches. It is also described how the sequence number scheme can be extended to deal with failures, such as the possibility of long delayed packets, network partitions and hardware failures.

Every broadcast started by a border node will be labeled with a higher sequence number than the one known so far. On receiving an update message with information that is more up-to-date than the information currently available, a node accepts this information by executing Algorithm II in Section 3.2. A node accepts a message as being more up-to-date, if the sequence number in the message is higher than the one kept until then. Nodes, therefore, maintain sequence numbers for every visible domain. Border nodes also have to keep sequence numbers for the domains of which they are border nodes. In order to keep this information up-to-date, every time an update of this number takes place, a broadcast of it is performed in the domain.

Using the same scenario as before, it can be shown that two broadcasts started by different border nodes for a domain D, labeled with the same sequence number, need not have the same value. However, in this case, at least one border node has not yet received at least one update

message of a broadcast for a subdomain of D currently in the network. It is guaranteed that this border node will receive the missing information within finite time and start a new broadcast for domain D. This new value may be the same value the other border node broadcasted for D. The same value may then be accepted twice by some nodes. The algorithm can be adapted to loose this inefficiency. Since optimizing our algorithm is beyond the scope of the work presented here, we do not deal with this issue.

We denote by $S_n(D)$ the sequence number of a domain D stored at node n. Initially for every node n, $S_n(D)$ and $m_n(D)$ contain 0 and $-\infty$, respectively, for every domain D of which n is part or which is visible to n. The set $Border(D)$ contains the border nodes of D. A broadcast of a value m_D for domain D labeled with sequence number S_D is done by sending $UPD(D, m_D, S_D)$ to all nodes in the superdomain of D.

Algorithm III (network, multiple changes)

- Upon detecting a change in $m_n(n)$ to m'_n, node n performs $\text{Accept}_n(n, m'_n, S_n(n) + 1)$.
- Upon receipt of $UPD(D, m_D, S_D)$, node n performs the following actions:

```
if S_D > S_n(D)
then if n ∈ D
     then S_n(D) := S_D;
          Broadcast_n(D, m_D, S_D)
              /* broadcast new sequence number of D */
     else Accept_n(D, m_D, S_D)
              /* update the m-value and
                 sequence number of D */
     fi
fi,
```

where

```
Broadcast_n(D, m_D, S_D) =
  Let D ◁ F in
  {for all {n, f}, f ∈ F
   do send UPD(D, m_D, S_D) to f
   od
  }

Accept_n(D, m_D, S_D) =
  Let D ◁ F in
  {m_n(D) := m_D;
   S_n(D) := S_D;
   Broadcast_n(D, m_D, S_D);
   if m_n(F) ≠ max{m_n(D_i) | D_i ◁ F}
   then if n ∈ Border(F)
        then Accept_n(F, max{m_n(D_i) | D_i ◁ F}, S_n(F) + 1)
                 /* accept new value for F, broadcast it and
                    check m-value of the superdomain of F */
        else Accept_n(F, max{m_n(D_i) | D_i ◁ F}, S_n(F))
                 /* accept new value for F and check m-value
                    of the superdomain of F */
        fi
   fi
  }
```

□

3.3.3 Correctness Proof

In this section we prove that Algorithm III is correct. This proof consists of two parts. First, the property that after the algorithm has terminated (i.e., no node is involved in a broadcast) all nodes have correct values is shown to hold. Next, it is proven that the algorithm terminates within finite time, after the last change in a node's input has occurred. Before proving correctness, some notation is introduced. Node n *accepts* $UPD(D, m_D, S_D)$ if as a consequence of the receipt of this message, $\text{Accept}_n(D, m_D, S_D)$ is executed. The algorithm *has terminated for domain* D if no update message containing a value of D exists in the network and the algorithm has terminated for every subdomain of D.

Lemma 3.1 *Suppose that the algorithm has terminated for domain D and for all nodes $d, d' \in D$: $m_d(C) = m_{d'}(C)$, for all $C \lhd D$. Then the following holds for nodes $n, n' \in F$, with $D \lhd F$:*

- $m_n(D) = \max\{m_n(C) \mid C \lhd D\}$, *if* $n \in D$.
- $m_n(D) = m_{n'}(D)$.
- $S_n(D) = S_{n'}(D)$.

Proof: From the algorithm it directly follows that $m_n(D) = \max\{m_n(C) \mid C \lhd D\}$, if $n \in D$. Therefore, especially after termination this equality holds. It now follows from the fact that all nodes in D hold the same values for the subdomains of D, that if $n, n' \in D$: $m_n(D) = \max\{m_n(C) \mid C \lhd D\} = \max\{m_{n'}(C) \mid C \lhd D\} = m_{n'}(D)$.

Let m_D be this final value all nodes in D have for D and $S_D = \max\{S_n(D) \mid n \in D\}$. All nodes increment their sequence number if there is a neighbour with a higher sequence number for the same domain. Only border nodes are able to increment their sequence number for D to start a broadcast. Therefore, $S_D = \max\{S_n(D) \mid n \in \text{Border}(D)\}$. Every (border) node, which started a broadcast labeled with S_D, must have sent $UPD(D, m_D, S_D)$ to neighbour-nodes within the superdomain F. If this is not the case, a node n received the value m_D later and started a broadcast labeled with $S_n(D) > S_D$, which is in contradiction with the assumption that S_D is the maximum sequence number for D.

Every node n receiving $UPD(D, m_D, S_D)$, accepts it unless already $S_n(D) = S_D$ (and $m_n(D) = m_D$, if $n \notin D$). From the connectiveness of every domain it easily follows that every node $n \in F$, receives at least once $UPD(D, m_D, S_D)$.

□

Lemma 3.2 *Suppose the algorithm has terminated for all domains $C \lhd D$. Then within finite time the algorithm has terminated for domain D.*

Proof: Let $S_D = \max\{S_n(D) \mid n \in \text{Border}(D)\}$ at the moment the algorithm has terminated for all $C \lhd D$. After the algorithm has terminated for domains $C \lhd D$, no node $n \in \text{Border}(D)$ will start a new broadcast for domain D. The last broadcast for domain D (labeled with highest sequence number) must be labeled with S_D. As before, from connectiveness of every domain, it follows that every node $n \in F$, must receive $UPD(D, m_D, S_D)$ at least once. After a node accepts $UPD(D, m_D, S_D)$, it will not accept any update message labeled with a number less than or equal to S_D. Therefore, all earlier broadcast messages will be "absorbed" within finite time (because of the assumption that messages are not on a link for an infinite amount of time).

□

Theorem 3.1 *Within finite time after the last $m_n(n)$ has changed:*

- *the algorithm terminates for the whole network and*
- *the nodes contain correct values for all domains they are supposed to have a value for.*

Proof: We prove by induction on the structure of the hierarchical division that within finite time the algorithm terminates for every domain D and every node in the superdomain of D has correct values for D.

Base: D is a node. Because D does not have any subdomain, this case directly follows from Lemmata 3.1 and 3.2.

Step: Suppose D is not a node. From the induction hypothesis, within finite time the algorithm terminates for every $C \lhd D$ and for all nodes $d, d' \in D : m_d(C) = m_{d'}(C)$. From Lemmata 3.1 and 3.2 the step follows.

□

4 Formal Description

In this Section we formalize the steps as given in Section 3. The theory UNITY, a computational model and a proof system, is used. We assume the reader to be familiar with the notation used for the programs and the logic for the specification, design, and verification of UNITY programs. This theory can be found in chapters 2 and 3 of [1].

4.1 Problem specification

Let G be a graph in which an *input* is associated with each node. Inputs of nodes may change spontaneously. It is required to compute and maintain the maximum of all the inputs. Here, we describe and derive in UNITY notation the algorithms as described in Section 3.

4.1.1 Notation

- The set $node(G)$ contains the nodes of the graph G.
- $\forall n \in node(G) : m(n)$ contains the input of node n.
- Variable $m(G)$ should contain the maximum of the inputs.
- Let $[a_1, a_2, \ldots, a_n]$ denote the *bag*, or *multiset*, of elements $a_1, a_2, \ldots, a_n$.
  If $n_1, \ldots, n_x$ are the nodes of G, then $nodesval(G) = [m(n_1), m(n_2), \ldots, m(n_x)]$.
- $\vec{M}$ denotes a bag of integers.
- When the algorithm has terminated with value v we say that $term(G, v)$ holds. Then, the variable $m(G)$ equals the maximum v of all inputs, i.e.,

$$term(G, v) \quad \Rightarrow \quad v = m(G) = \langle \max n : n \in node(G) :: m(n) \rangle.$$

4.1.2 Specification

The specification of maintaining and obtaining the maximum value of the inputs in a network G, where each node has an input, can be formulated as follows:

SP1: $term(G, v) \wedge nodesval(G) = \vec{M}$ unless $nodesval(G) \neq \vec{M}$

SP2: $nodesval(G) = \vec{M} \mapsto \exists v : term(G, v) \vee nodesval(G) \neq \vec{M}$

4.2 Maximum computation on a division tree

Upon the graph G a hierarchical structure is imposed, which can be modeled as a division tree. Let the inputs of the graph G be placed as values at the leaves of this tree. By computing (i.e. obtaining and maintaining) at every internal node a value which is the maximum of the values at its sons, the root node of the tree obtains and maintains the maximum value of the inputs in G.

4.2.1 Notation

For every node D (of the tree):

- The set $leaf(D)$ contains D itself if D is a leaf or else the leaves under the node D.
- $m(D)$ is D's view of the maximum of the inputs under D.
- $m_D(C)$ is D's view of $m(C)$ for $C \lhd D$.
- Let $+\!\!+$ be the *bag-union*. Then $leavesval(D)$ is defined as follows:
  $leavesval(D) = [m(D)]$, if $D \in leaf(D)$.
  Otherwise, let $C_1, \ldots, C_x$ be the sons of D, then
  $leavesval(D) = leavesval(C_1) +\!\!+ leavesval(C_2) +\!\!+ \ldots +\!\!+ leavesval(C_x)$.
  Note: $leavesval(R) \equiv nodesval(G)$, if R is the root of the division tree corresponding with the hierarchical division on G.
- The value of a node D must be made known to its father. $fatherknows(D, v)$ is true if the father of D has a correct view of $m(D)$.

$$fatherknows(D, v) \equiv \langle \wedge F : D \lhd F :: m_F(D) = v \rangle.$$

  If R is the root of the division tree, we define $fatherknows(R, v) \equiv true$.
- The algorithm is *finished* in some internal node D if the variable $m(D)$ contains a value v that is the maximum of all the values of its sons, and the algorithm has terminated (which is defined next) in these sons.

$$hfinished(D, v) \equiv \begin{cases} v = m(D) & \text{if } D \in leaf(D) \\ v = m(D) = \langle \max C : C \lhd D :: m_D(C) \rangle \wedge & \\ \qquad \forall C \lhd D : hterm(C, m(C)) & \text{otherwise.} \end{cases}$$

- The algorithm has *terminated* in some internal node D if the algorithm is finished in D and the value of D is known by the father of D.

$$hterm(D, v) \equiv hfinished(D, v) \wedge fatherknows(D, v).$$

  Note: $hterm(R, v) \equiv hfinished(R, v)$, if R is the root of the division tree.

4.2.2 The Solution Strategy: Formal Description

The specification of obtaining and maintaining the maximum value of the inputs at the leaves under a node D in a division tree, can be formally formulated as follows:

SP3: $hfinished(D, v) \wedge leavesval(D) = \vec{M}$ unless $leavesval(D) \neq \vec{M}$

SP4: $hterm(D, v)$ unless $\neg hfinished(D, v)$

SP5: $\forall C \lhd D : hterm(C, v_c) \mapsto (\exists v : hfinished(D, v)) \vee (\exists C : \neg hfinished(C, v_c))$

SP6: $hfinished(D, v) \mapsto hterm(D, v) \vee \neg hfinished(D, v)$

4.2.3 Proof of Correctness of the Solution Strategy

We prove that the unless-relation (SP1) and the progress-condition (SP2), given in the problem specification (in Section 4.2.1), are met by any strategy that satisfies conditions SP3, SP4, SP5 and SP6.

The proof that SP1 is met, follows directly from Lemma 4.1 which is presented next.

Lemma 4.1 $hterm(D,v) \equiv v = m(D) = \langle \max n : n \in leaf(D) :: m(n) \rangle \wedge fatherknows(D,v) \wedge \forall C \lhd D : hterm(C, m(C))$.

Proof: To prove the lemma, we use induction on the structure of the division-tree.

Base: D is a leaf. From the definition of $hterm(D,v)$, with D a leaf of the division-tree, this case directly follows.

Step: Suppose D is not a leaf and the lemma holds for all sons of D, i.e., $\forall C \lhd D : [hterm(C, v_c) \equiv v_c = m(C) = \langle \max n : n \in leaf(C) :: m(n) \rangle \wedge fatherknows(C, v_c) \wedge \forall B \lhd C : hterm(B, m(B))]$.

Then the following holds:

$$
\begin{aligned}
& hterm(D,v) \\
& \quad \equiv hfinished(D,v) \wedge fatherknows(D,v) \\
& \quad \equiv v = m(D) = \langle \max C : C \lhd D :: m_D(C) \rangle \wedge \forall C \lhd D : hterm(C, m(C)) \wedge \\
& \qquad fatherknows(D,v) \\
& \quad \{hterm(C, m(C)) \Rightarrow (m_D(C) = m(C))\} \\
& \quad \equiv v = m(D) = \langle \max C : C \lhd D :: m(C) \rangle \wedge \forall C \lhd D : hterm(C, m(C)) \wedge \\
& \qquad fatherknows(D,v) \\
& \quad \{\text{Induction hypothesis}\} \\
& \quad \equiv v = m(D) = \langle \max C : C \lhd D :: \langle \max n : n \in leaf(C) :: m(n) \rangle \rangle \wedge \\
& \qquad \forall C \lhd D : hterm(C, m(C)) \wedge fatherknows(D,v) \\
& \quad \equiv v = m(D) = \langle \max n : n \in leaf(D) :: m(n) \rangle \wedge \forall C \lhd D : hterm(C, m(C)) \wedge \\
& \qquad fatherknows(D,v).
\end{aligned}
$$

This proves the induction step.

We can conclude that $hterm(D,v) \equiv v = m(D) = \langle \max n : n \in leaf(D) :: m(n) \rangle \wedge \forall C \lhd D : hterm(C, m(C)) \wedge fatherknows(D,v)$.

□

Corollary 4.1 *If R is the root of the division tree corresponding to a hierarchical division on G, it is possible to define: $hterm(R,v) \equiv term(G,v)$.*

In the following we will use this definition. SP1 directly follows from this and SP3. We will now prove that SP2 is met.

Lemma 4.2

$$
\begin{aligned}
& (\forall C \lhd D : (hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c)) \mapsto \\
& \qquad (\forall C \lhd D : hterm(C, v_c)) \vee \neg(\forall C \lhd D : leavesval(C) = \vec{M}_c).
\end{aligned}
$$

Proof:

PSP theorem on SP3 and SP6:

$(hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c)) \mapsto$
$[(hterm(C, v_c) \vee \neg hfinished(C, v_c)) \wedge hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c]$
$\vee \neg(leavesval(C) = \vec{M}_c)$.

Rewriting the right side:

$(hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c)) \mapsto$
$[hterm(C, v_c) \wedge leavesval(C) = \vec{M}_c] \vee \neg(leavesval(C) = \vec{M}_c)$.

Rewriting the right side:

$(hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c)) \mapsto$
$hterm(C, v_c) \vee \neg(leavesval(C) = \vec{M}_c)$.

Conjunction over the $C's$ on the above:

$\forall C \lhd D : [(hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c)) \mapsto$
$hterm(C, v_c) \vee \neg(leavesval(C) = \vec{M}_c)]$.

From SP4:

$\forall C \lhd D : [(hterm(C, v_c)$ unless $\neg(hfinished(C, v_C))]$.

The generalization theorem of the completion theorem on the above two:

$\forall C \lhd D : (hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c) \mapsto$
$(\forall C \lhd D : hterm(C, v_C)) \vee \neg(\forall C \lhd D : leavesval(C) = \vec{M}_c \wedge hfinished(C, v_c))$.

From SP3, using simple conjunction:

$\forall C \lhd D : (hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c)$ unless $\neg(\forall C \lhd D : leavesval(C) = \vec{M}_c)$.

The PSP theorem on the above two:

$\forall C \lhd D : (hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c) \mapsto$
$[(\forall C \lhd D : hterm(C, v_C) \vee \neg(\forall C \lhd D : leavesval(C) = \vec{M}_c \wedge hfinished(C, v_c)))$
$\wedge(\forall C \lhd D : leavesval(C) = \vec{M}_c \wedge hfinished(C, v_c))]$
$\vee\neg(\forall C \lhd D : leavesval(C) = \vec{M}_c)$.

Rewriting the right side:

$\forall C \lhd D : (hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c) \mapsto$
$\forall C \lhd D : [hterm(C, v_C) \wedge leavesval(C) = \vec{M}_c \wedge hfinished(C, v_c)]$
$\vee\neg(\forall C \lhd D : leavesval(C) = \vec{M}_c)$.

Weakening the right side:

$\forall C \lhd D : (hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c) \mapsto$
$(\forall C \lhd D : hterm(C, v_C)) \vee \neg(\forall C \lhd D : leavesval(C) = \vec{M}_c)$.

□

Lemma 4.3

$$\forall C \lhd D : (hterm(C, v_c) \wedge leavesval(C) = \vec{M}_c) \mapsto$$
$$(\exists v : hfinished(D, v)) \vee \neg(\forall C \lhd D : leavesval(C) = \vec{M}_c).$$

Proof:

From SP3, using simple conjunction:

$\forall C \lhd D : (hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c)$ unless $\neg(\forall C \lhd D : leavesval(C) = \vec{M}_c)$.

The PSP theorem on SP5 and the above:

$\forall C \lhd D : (hterm(C, v_c) \wedge hfinished(C, v_C) \wedge leavesval(C) = \vec{M}_c) \mapsto$
$[(\exists v : hfinished(D, v) \vee \neg(\forall C \lhd D : hfinished(C, v_C)))$
$\wedge\forall C \lhd D : (hfinished(C, v_C) \wedge leavesval(C) = \vec{M}_c)]$
$\vee\neg(\forall C \lhd D : leavesval(C) = \vec{M}_c)$.

Rewriting the left- and right side:

$\forall C \lhd D : (hterm(C, v_c) \wedge leavesval(C) = \vec{M}_c) \mapsto$
$[\exists v : hfinished(D, v) \wedge \forall C \lhd D : (hfinished(C, v_C) \wedge leavesval(C) = \vec{M}_c)]$
$\vee\neg(\forall C \lhd D : leavesval(C) = \vec{M}_c)$.

Weakening the right side:

$\forall C \lhd D : (hterm(C, v_c) \wedge leavesval(C) = \vec{M}_c) \mapsto$
$(\exists v : hfinished(D, v)) \vee \neg(\forall C \lhd D : leavesval(C) = \vec{M}_c)$.

□

Lemma 4.4 *For all domains D:*

$$(leavesval(D) = \vec{M}) \mapsto \exists v : hfinished(D, v) \vee \neg(leavesval(D) = \vec{M}).$$

Proof: We will give a prove by induction on the structure of the division-tree.
Base: If D is a leaf, this case trivially follows.
Step: Suppose D is an internal node.

From Induction Hypothesis:
$\forall C \lhd D : [(leavesval(C) = \vec{M}_c) \mapsto \exists v_c : hfinished(C, v_c) \vee \neg(leavesval(C) = \vec{M}_c)]$.
Rewriting the right side:
$\forall C \lhd D : [(leavesval(C) = \vec{M}_c) \mapsto$
$(\exists v_c : hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c) \vee \neg(leavesval(C) = \vec{M}_c)]$.
From SP3:
$\forall C \lhd D : [(hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c)$ unless $\neg(leavesval(C) = \vec{M}_c)]$.
Using the generalization of the completion theorem on the above two relations:
$(\forall C \lhd D : leavesval(C) = \vec{M}_c) \mapsto$
$[\forall C \lhd D : (\exists v_c : hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c)$
$\vee \neg(\forall C \lhd D : leavesval(C) = \vec{M}_c)]$.
The cancellation theorem on the result of Lemma 4.2 and the above:
$(\forall C \lhd D : leavesval(C) = \vec{M}_c) \mapsto$
$[\forall C \lhd D : (\exists v_c : hterm(C, v_c)) \vee \neg(\forall C \lhd D : leavesval(C) = \vec{M}_c)]$.
Rewriting the right side:
$(\forall C \lhd D : leavesval(C) = \vec{M}_c) \mapsto$
$\forall C \lhd D : (\exists v_c : hterm(C, v_c) \wedge leavesval(C) = \vec{M}_c)$
$\vee \neg(\forall C \lhd D : leavesval(C) = \vec{M}_c)$.
Using transitivity on the result of Lemma 4.3 and the above:
$(\forall C \lhd D : leavesval(C) = \vec{M}_c) \mapsto$
$(\exists v : hfinished(D, v)) \vee \neg(\forall C \lhd D : leavesval(C) = \vec{M}_c)$.
Rewriting the left- and right side concludes the induction step, thereby proving the lemma.

□

Theorem 4.1 *SP2 is met by SP3, SP4, SP5 and SP6.*

Proof: Let R be the root of the division tree, then from Lemma 4.4 and corollary 4.1:

$$(leavesval(R) = \vec{M}) \mapsto \exists v : hterm(R, v) \vee \neg(leavesval(R) = \vec{M}).$$

Because $(leavesval(R) \equiv nodesval(G))$ and $(hterm(R, v) \equiv term(G, v))$, SP2 follows.
□

4.2.4 Derivation of a Program from the Specification

A program follows directly from the specification of the solution strategy. The function $inp(n)$ delivers initially the input of some leaf n. The possibility of the alteration of an input $m(n)$, of some leaf n, is implemented by the assignment $m(n) := g(n)$.

Program {maximum computation on a division tree}
initially
$\langle \| D ::$
$m(D) = inp(D)$ if $D \in leaf(D) \sim -\infty$ if $\neg(D \in leaf(D))$
$\|$
$\langle \| C : C \lhd D :: m_D(C) = -\infty \rangle$
$\rangle$

assign

$\langle |D ::$

$m(D) := g(D)$ if $D \in leaf(D) \sim$

$\langle \max C : C \lhd D :: m_D(C) \rangle$ if $\neg(D \in leaf(D))$

$|$

$\langle |C : C \lhd D :: m_D(C) := m(C) \rangle$

$\rangle$

end {maximum computation on a division tree}

4.3 Maximum computation on a divided graph

We now derive the distributed algorithm on a hierarchically divided network.

If D is an internal node of the division tree corresponding with the hierarchical division on the network, D corresponds with a domain in this network, and vice versa. Therefore in the following D may denote a node in the division tree or a domain in the hierarchically divided network. From the context it will always be clear what is meant.

4.3.1 Notation

For every domain D:

- The set $leaf(D)$ contains D itself if D is a node or else the nodes in domain D.
- $m_n(D)$ contains the value of D stored at node n.
- $S_n(D)$ contains the sequence-number of D stored at node n.
- Let $+\!\!+$ be the *bag-union*. Then $Dleavesval(D)$ is defined as follows:
  $Dleavesval(D) = [m_D(D)]$, if $D \in leaf(D)$.
  Otherwise, let $C_1, \ldots, C_x$ be the subdomains of D, then
  $Dleavesval(D) = Dleavesval(C_1) +\!\!+ Dleavesval(C_2) +\!\!+ \ldots +\!\!+ Dleavesval(C_x)$.
  Note: $Dleavesval(D) \equiv leavesval(D)$
- $Dfatherknows(D, v)$ holds iff all nodes in the superdomain of D has v as value for D, i.e.,

$$Dfatherknows(D, v) \equiv \langle \wedge n : n \in F \wedge D \lhd F :: m_n(D) = v \rangle$$

- Let $n \in D$, n is ready for D iff the value v, as stored by n for D, equals the maximum of the values of the subdomains as kept by n.

$$Nfinished(n, D, v) \equiv \begin{cases} v = m_n(D) & \text{if } D \in leaf(D) \\ v = m_n(D) = \langle \max C : C \lhd D :: m_n(C) \rangle & \text{otherwise} \end{cases}$$

- $Dhfinished(D, v)$ holds iff all nodes are ready for D with value v and all subdomains have terminated (which is defined next), i.e.,

$$Dhfinished(D, v) \equiv \langle \wedge n : n \in leaf(D) :: Nfinished(n, D, v) \rangle \wedge \forall C \lhd D : (\exists v_c : Dhterm(C, v_c))$$

- $Dhterm(D, v)$ holds iff domain D has terminated with value v, that is all nodes in D are ready for D with value v and all nodes in the superdomain of D has v as value for D.

$$Dhterm(D, v) \equiv Dhfinished(D, v) \wedge Dfatherknows(D, v)$$

-

$$equal(F, D, \overline{S}) \equiv \langle \wedge n : n \in leaf(F) \wedge D \lhd F :: S_n(D) = \overline{S} \rangle$$

•

$$maxs(D,\overline{S}) \equiv \overline{S} = \langle \max n : D \lhd F \wedge n \in leaf(F) :: S_n(D) \rangle$$

4.3.2 The Solution Strategy: Formal Description

The specification of obtaining and maintaining the maximum value of the inputs at the nodes in a domain D by all the nodes in D, can be formally formulated as follows:

SP7: $Dhfinished(D,v) \wedge Dleavesval(D) = \vec{M}$ unless $Dleavesval(D) \neq \vec{M}$

SP8: $Dhterm(D,v)$ unless $\neg Dhfinished(D,v)$

SP9: $\forall C \lhd D : Dhterm(C,v_c) \mapsto (\exists v : Dhfinished(D,v)) \vee (\exists C \lhd D : \neg Dhfinished(C,v_c))$

SP10: $Dhfinished(D,v) \wedge maxs(D,\overline{S}) \mapsto equal(F,D,\overline{S}) \vee \neg Dhfinished(D,v)$

SP11: **Invariant**
$Dhfinished(D,v) \wedge equal(F,D,\overline{S}) \Rightarrow Dhterm(D,v)$

4.3.3 Proof of Correctness of the Solution Strategy

By defining,

$$(m_D(C) = v) \Leftrightarrow \langle \wedge n : n \in leaf(D) :: m_n(C) = v \rangle$$

the following equalities hold:

$$\begin{aligned} fatherknows(D,v) &\equiv Dfatherknows(D,v) \\ hfinished(D,v) &\equiv Dhfinished(D,v) \\ hterm(D,v) &\equiv Dhterm(D,v) \end{aligned}$$

From above SP3, SP4 and SP5 directly follow from SP7, SP8 and SP9 respectively.

We will now prove that SP6 is met by SP10 and SP11. Note that there is always a $\overline{S}$ such that $maxs(D,\overline{S})$ holds.

Suppose $maxs(D,\overline{S})$, then from the implication theorem:
$Dhfinished(D,v) \mapsto Dhfinished(D,v) \wedge maxs(D,\overline{S})$
Transitivity on the above and SP10:
$Dhfinished(D,v) \mapsto equal(F,D,\overline{S}) \vee \neg Dhfinished(D,v)$
Rewriting the right side:
$Dhfinished(D,v) \mapsto (equal(F,D,\overline{S}) \wedge Dhfinished(D,v)) \vee \neg Dhfinished(D,v)$
SP11:
$Dhfinished(D,v) \mapsto Dhterm(D,v) \vee \neg Dhfinished(D,v)$

From above and the observations made SP6 follows.

4.3.4 Derivation of a Program from the Specification

The set E contains all links existing in the network. A link between nodes n and b is denoted by $\{n,b\}$.

Program {maximum computation on a divided network using shared memory}
initially
$\langle \mathbf{I}n, D, F : n \in F \wedge D \lhd F :: m_n(D), S_n(D) = inp(n),1 \quad \text{if } n = D \sim$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad -\infty, \quad 0 \quad \text{if } \neg(n = D)$
$\rangle$

assign

$\langle \| n ::$

$\langle \| D, F : n \in D \wedge D \lhd F ::$

$m_n(D), S_n(D) := g(n), S_n(D) + 1 \quad$ if $n = D \sim$

$\langle \max C : C \lhd D :: m_n(C) \rangle, S_n(D) + 1$

if $((m_n(D) \neq \langle \max C : C \lhd D :: m_n(C) \rangle)$

$\wedge (n \neq D))$

$\| \langle \| b : b \in F \wedge \{n, b\} \in E :: S_n(D) := S_b(D) \quad$ if $(S_b(D) > S_n(D))$

$\rangle$

$\rangle$

$\|$

$\langle \| D, F : \neg(n \in D) \wedge (D \lhd F) \wedge (n \in F) ::$

$\langle \| b : b \in F \wedge \{n, b\} \in E :: m_n(D), S_n(D) := m_b(D), S_b(D) \quad$ if $(S_b(D) > S_n(D))$

$\rangle$

$\rangle$

$\rangle$

end {maximum computation on a divided network using shared memory }

The proof that the program satisfies the solution strategy is rather straightforward. SP7, SP8, SP9 and SP10 follow directly from the program text. SP11 follows from the following invariant, which can be easily proven from the program text. In the following variables i and j range over the set $\{1, \ldots, x\}$, with $x > 1$.

invariant

$[(n \in F) \wedge (n \notin D) \wedge (D \lhd F)] \Rightarrow$

$\langle \exists n_0, \ldots, n_x :$

$n_0 \in D \wedge \langle \wedge n_i n_j :: n_i \notin D \wedge n_i \in F \wedge ((i \neq j) \Rightarrow (n_i \neq n_j)) \rangle \wedge n_x = n ::$

$[(S_{n_i}(D) = S_{n_{i-1}}(D) \wedge m_{n_i}(D) = m_{n_{i-1}}(D)) \vee (S_{n_i}(D) < S_{n_{i-1}}(D))]\rangle$

This program can be implemented on a distributed system in the obvious way. The variables $m_n(D)$ and $S_n(D)$ are then local to n. A change in one of these variables is made known to the neighbours of n, which are in the superdomain of D, by sending it over the channels to these neighbours.

5 Conclusion

In this paper we derived a distributed algorithm on a hierarchically divided network. Every node in the network has an associated input-value and a variable in which it keeps the maximum of all inputs. These inputs may change spontaneously. Within finite time after a change, every node has updated its maximum value of the inputs.

The algorithm is developed in stages. After the problem description, an abstract algorithm on a (virtual) division tree is given. Next, the algorithm is transformed in the distributed algorithm that operates on a hierarchically divided network.

The stages are described informally as well as in UNITY. This combination appeared to be an ideal combination to design distributed and concurrent algorithms. It is our hope that the derivation and techniques used in the algorithm will be helpful in the future to design more distributed algorithms on hierarchically divided networks.

6 Acknowledgement

The authors thank Dr. G. Tel for his help during the course of this research.

References

[1] K. Mani Chandy and Jayadev Misra. *Parallel Program Design – A Foundation.* Addison-Wesley Publishing Company, Inc., 1988.

[2] Leslie Lamport. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming*, (2):175–206, 1982.

[3] P.J.A. Lentfert, A.H. Uittenbogaard, S.D. Swierstra, and G. Tel. Distributed hierarchical routing. Technical Report RUU-CS-89-5, Utrecht University, March 1989; also pp. 321-344 in *Proceedings Computing Science in the Netherlands CSN'89*, SION, Utrecht (1989).

[4] R. Perlman. Fault-tolerant broadcast of routing information. *Computer Networks*, 7:395–405, December 1983.

[5] Adrian Segall. Distributed network protocols. *IEEE Trans. Inf. Theory*, IT-29(1):23–35, January 1983.

Distributed incremental maximum finding in hierarchically divided graphs

P.J.A. Lentfert, S.D. Swierstra, A.H. Uittenbogaard

Technical Report RUU-CS-90-30
September 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN:0924-3275

Distributed Incremental Maximum Finding in Hierarchically Divided Graphs

P.J.A. Lentfert, S.D. Swierstra
Department of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht,
The Netherlands.

A.H. Uittenbogaard
HCS Industrial Automation B.V.
P.O. Box 20020, 7302 HA Apeldoorn,
The Netherlands.

1 Introduction

Distributed algorithms are often informally described. An advantage of an informal description is the seeming ease with which they can be produced and studied. It is not necessary to understand a formal system, before finding out the ideas behind the algorithm.

However, a major disadvantage in providing only informal arguments is that errors are easily made. It is our experience that distributed algorithms which seemed to be correct, and were even informally proven to be correct, in fact were wrong.

For example, during our research ([3]), we once were convinced to have designed a correct distributed algorithm. By coincidence, we noticed that in some cases the update algorithm did not terminate. In our proof we showed that the algorithm would terminate after any update, by presenting a function whose value decreased in every step of the algorithm and was bounded from below. Unfortunately, this function could have an infinite value, in which case the lower bound would never be reached.

It was decided to use a formal method to be more sure of a correct outcome. UNITY ([1]) was chosen, a formalism which is claimed to be helpful in the design of distributed and concurrent programs. In [1], Hoare claims that UNITY is a complete theory of programming because it includes the following important aspects:

- It is a method for specification of programs.
- It is a method for reasoning about specifications.
- It is a method of developing programs together with a proof that they meet their specification.
- It is a method of transforming programs to achieve high efficiency on the machines available for their execution.

The design and study of algorithms using a formalism can be extremely tiresome, although rewarding. It is our experience that formal methods (including UNITY) soon become very complex when studying nontrivial problems. Therefore, an informal description may be very helpful.

In [3] we introduced the distributed hierarchical routing algorithm that operates on a hierarchically divided network, i.e., a network that is divided in domains and each domain, in turn, is divided in subdomains, etc. As a result of hierarchically dividing the network, the size of the routing tables remains small. The size of the tables for nodes in a hierarchically divided network containing N nodes can be reduced to $O(\log N)$ entries.

It is our goal to formally design a distributed update algorithm on a hierarchically divided network upon which the distributed hierarchical routing algorithm operates. Before using UNITY on our update algorithm, we have decided to first gain some experience in using UNITY on a simpler problem. This algorithm to be derived is used to obtain and maintain the maximum value of a set of frequently changing inputs. Every node in a hierarchically divided network is associated with precisely one element of this set of inputs, and every node has to maintain the maximum value of the inputs of all nodes.

This problem and the problem of keeping topological information about a hierarchically divided network bear some similarities. The network can be considered as a set of nodes. With each node in this set corresponds a possibly changing set of paths from a source node to that node. Every node must, for every other destination node in the set, obtain and maintain the best path of the set of paths to that destination node, according to some metric (for example the hop-count).

The problem of maintaining this routing information is for two reasons more complex than the problem of keeping the maximum value. In the first place, it deals with a set of sets, viz., for each node in the set of destination nodes there is a set of paths to it.

The second reason is that every node has to deal with a different set of paths for the same destination node. In keeping the maximum value of a set of inputs, each node has to deal with one, and the same, set of integers.

We are, however, convinced that the (formal) derivation of a distributed algorithm for keeping the maximum value of a frequently changing set of inputs, that operates on a hierarchically divided network, will help us in the (formal) derivation of an update algorithm.

In Section 2 the topological model is described. The description of the algorithm consists of two parts. Because, as stated above, an informal description may be very helpful, we have included in Section 3, a traditional description and solution of the problem. In that part we develop an algorithm by first designing an abstract algorithm that operates on a virtual datastructure. In the next step, this algorithm is transformed into a distributed algorithm that operates on a hierarchically divided network. Also a more traditional proof of the distributed algorithm is given.

In Section 4, these steps are formalized.

2 Model

In this article we describe a distributed algorithm on a network that is identified with a bidirectional graph, the *network graph*. The nodes of this graph represent processors in the network, the edges represent communication links between them. Every node has local computing capacity and memory to which no other node has (direct) access. Nodes cooperate by exchanging messages over the links connecting them. These links are assumed to be fault-free and have a FIFO behaviour.

Upon the graph a hierarchical structure is imposed ([3]): nodes are grouped into domains, which, in turn, are grouped into superdomains, etc. Nodes are viewed as the smallest possible domains. The graph, thus divided, is called a *hierarchically divided graph*. The division of the graph is modeled as a tree, called the *division tree*. The network domain (the domain comprising the whole network) corresponds to the root of this tree. Subdomains of a domain correspond to sons of the tree-node corresponding to the domain. The leaves of the division tree correspond to the processors in the network.

When tree-node m in the division tree is a son of tree-node n, we write: $m \lhd n$. Analogously, when domain C in the hierarchically divided network is a subdomain of domain D, we write: $C \lhd D$. Tree-node n in the division tree is *visible* to tree-node m, if n is the son of a tree-node on the path in the division tree to m, but is not on that path itself. Analogously, domain D in the hierarchically divided network is visible to domain C if D is a direct subdomain of a domain E of which C is a descendant, too, and C is not contained in D.

Note that, by definition, if a domain D is visible to domain C, then D is visible to every node in C. Also, subdomains of a domain that is visible to a node are not visible to that node. As a consequence, internal details of a domain are hidden from domains, especially from nodes, outside that domain. This will result in a reduction of the storage complexity of the algorithms to be

developed: only a single item of information has to be stored at a node n for each domain visible to n, and not for every single node in it.

On the hierarchical division we impose the following restriction: every domain is connected. In other words: a message routed between two nodes does not have to leave the least common domain of those nodes. In order to meet this constraint, it is possible, as described in [3], that subdomains are shared between domains, i.e., domains may be overlapping.

3 Traditional Description

3.1 Problem

In this section we describe the problem that is solved in the rest of this article.

We assume that every node in a hierarchically divided network has an associated data item, called its *input*. The inputs are, not necessarily unique, elements from a totally ordered set. Furthermore, every node possesses a variable in which it keeps the maximum of all inputs in the network.

Inputs of nodes may change spontaneously, and as a result the global maximum may change. If this is the case, all nodes in the network are required to have updated their maximum value within finite time after the update has occurred. If multiple changes occur, nodes must have updated their maximum value within finite time after the last update.

3.2 Update Algorithm on a Division Tree

In this section we present an update algorithm that operates on a division tree after the input of a leaf has changed. In this algorithm, leaves as well as internal tree-nodes of the division tree are assumed to be capable of performing actions and of storing values (of variables). Keep in mind, however, that in a physical network, nodes are the only entities having these properties. Since internal tree-nodes in the division tree model domains in the hierarchical division of the network, the actions of the domains have to be emulated by the physical nodes "on behalf of the domains."

The following questions, then, have to be answered. Which nodes simulate the actions corresponding to a domain? Do we need special nodes, or can the domain actions be established by all nodes (in the domain) constituting a part in it?

In Section 3.3 we will describe a transformation of the abstract algorithm, to an algorithm in which the only active entities are nodes in a hierarchically divided network.

Every tree-node n in the division tree maintains the following variables:

$f(n)$: the father of tree-node n in the division tree,

$m(n)$: the *m-value* of n, defined as the input of n if n is a leaf, and (an approximation of) $\max\{m(s) \mid s \lhd n\}$ if n is an internal tree-node, and

$m_n(s)$: the current estimate of $m(s)$, for every son s of n, if n is an internal tree-node.

We assume a correctly initialized datastructure, i.e., every leaf n has got an input $m(n)$, and for every internal tree-node n, $m(n) = \max\{m_n(s) \mid s \lhd n\}$ and for every son s of n, $m_n(s) = m(s)$. Thus, $m(r)$ contains the maximum value of all leaves in the tree if r is the root of the division tree.

We now describe the distributed algorithm that is performed as a result of a change in the input of a leaf. The update algorithm is straightforward. Upon detecting a change in $m(n)$, tree-node n reports this change to $f(n)$. In response, $f(n)$ updates its value and, if it changes, recursively, reports this to $f(f(n))$.

The resulting algorithm is presented slightly more formally below.

Algorithm I (division tree)

- Upon detecting a change in $m(n)$ to m', leaf n sends message $UPD(n, m')$ to $f(n)$.
- Upon receipt of $UPD(s, m')$, internal tree-node n performs the following actions:

  $m_n(s) := m'$;
  $m(n) := \max\{m_n(s_i) \mid s_i \lhd n\}$;
  if $m(n)$ changed and n is not the root
  then send $UPD(n, m(n))$ to $f(n)$
  fi

□

3.3 Update Algorithm on a Hierarchically Divided Graph

In this section we will distribute the algorithm on a division tree as described in Section 3.2. A domain (an internal tree-node in the division tree) is simulated by the cooperation of all nodes contained in it. For that purpose we distribute the information kept by any internal tree-node D in the division tree in Algorithm I over the nodes contained in domain D in the hierarchically divided network. This is realized by a "projection" in the division tree of all information onto every leaf in the subtree below the tree-node containing the information. As a result, every node keeps the m-values of its superdomains *and* the m-values of its visible domains.

We denote by $m_n(D)$ the m-value of domain D stored at node n. In a stable state (for example before any changes occur) all nodes in domain D will have the same value for $m_n(D)$.

In Section 3.3.1 we handle the update of data after a single change has occurred. In Section 3.3.2 we extend the algorithm to perform the necessary actions after multiple changes have occurred "at the same time." In Section 3.3.3 this algorithm is proven to be correct.

3.3.1 Updating a Single Change

In the following description, let n be a node with direct superdomain D, which has direct superdomain F (i.e., $n \lhd D \lhd F$). If n's input changes to m'_n, every brother of n, i.e., every node d in D, has to update its value $m_d(n)$. This is established by broadcasting within D the message $UPD(n, m'_n)$, using Propagation of Information, see [5].

The change of n's input may affect the m-value of D. For example, if n's input was the maximal input in D, and it decreased, D's m-value decreases too (unless another node in D has this same maximal input). An increase of n's input on the other hand, may result in an increase in D's m-value. Anyway, if D's m-value changes, the new m-value can be determined by every node d inside D by computing $m_d(D) = \max\{m_d(n_i) \mid n_i \lhd D\}$. This changed value has to be reported to all other nodes in F. This is established by a broadcast of $UPD(D, m'_D)$, where m'_D is the new m-value of D. Nodes in D with neighbour-nodes outside D but in F, called *border nodes* of D, start broadcasting the new m-value.

Analogously, this change in D may affect F's m-value, necessitating the same actions of the nodes in F and in the direct superdomain of F. This procedure is repeated on every level for which the change has an effect.

The resulting algorithm is presented below. The notation $\{n, n'\}$ is used to denote a link between nodes n and n'.

Algorithm II (network, single change)

- Upon detecting a change in $m(n)$ to m'_n, node n performs $\text{Accept}_n(n, m'_n)$.
- Upon receipt of $UPD(D, m_D)$, node n performs the following actions:

  if $m_n(D) \neq m_D$
  then $\text{Accept}_n(D, m_D)$
  fi,

where

$\text{Accept}_n(D, m_D) =$
Let $D \lhd F$ **in**
$\{m_n(D) := m_D;$
for all $l = \{n, f\}, f \notin D \wedge f \in F$
do send $UPD(D, m_D)$ to f
od;
if $m_n(F) \neq \max\{m_n(D_i) \mid D_i \lhd F\}$
then /* new round necessary */
$\text{Accept}_n(F, \max\{m_n(D_i) \mid D_i \lhd F\})$
fi
$\}$

□

Note, that the algorithm requires at most two update messages to be sent over every link in the network. Moreover, the update messages are confined to the lowest level domain with a subdomain of which the input changes.

3.3.2 Updating After Multiple Changes

In this section we extend Algorithm II to deal with the case of multiple changes of inputs occurring at the same time. When the input of a node changes, the node starts reporting this change using Algorithm II. It is possible, however, that during this process another node changes its input, too. Since the information about this change will not immediately reach the first node, this node may start a broadcast based on out-of-date information. Therefore, update messages reporting different m-values of the same domain may be in transit at the same time.

As before, in order to simulate the actions of a domain by the cooperation of the nodes in it, every node in the domain maintains the m-value of each of its subdomains. Because of this projection of all information in the division tree onto every leaf, nodes are not aware of the internal structure of their visible domains. Specifically, nodes do not keep the m-values of subdomains of their visible domains. Nodes, therefore, are *not* able to determine whether the information received from a visible domain is up-to-date or not (i.e., whether the value equals the maximum of the inputs of the subdomains).

To distinguish between succeeding broadcasts for a domain we introduce a sequence number ([5]) per domain. In [4], Perlman describes a broadcast algorithm of routing information that stabilizes in reasonable time without human intervention after any malfunctioning equipment is repaired or disconnected. It is shown why in a network without globally synchronized clocks the sequence number scheme is better suited than local clocks for determining whether a received packet is older or newer than the last received packet. The problems with sequence numbers, such as the possibility of wrap-around, must also be considered using logical clocks. However, the logical clock scheme requires hardware that is not always available in packet switches. It is also described how the sequence number scheme can be extended to deal with failures, such as the possibility of long delayed packets, network partitions and hardware failures.

Every broadcast started by a border node will be labeled with a higher sequence number than the one known so far. On receiving an update message with information that is more up-to-date than the information currently available, a node accepts this information by executing Algorithm II in Section 3.2. A node accepts a message as being more up-to-date, if the sequence number in the message is higher than the one kept until then. Nodes, therefore, maintain sequence numbers for every visible domain. Border nodes also have to keep sequence numbers for the domains of which they are border nodes. In order to keep this information up-to-date, every time an update of this number takes place, a broadcast of it is performed in the domain.

Using the same scenario as before, it can be shown that two broadcasts started by different border nodes for a domain D, labeled with the same sequence number, need not have the same value. However, in this case, at least one border node has not yet received at least one update

message of a broadcast for a subdomain of D currently in the network. It is guaranteed that this border node will receive the missing information within finite time and start a new broadcast for domain D. This new value may be the same value the other border node broadcasted for D. The same value may then be accepted twice by some nodes. The algorithm can be adapted to loose this inefficiency. Since optimizing our algorithm is beyond the scope of the work presented here, we do not deal with this issue.

We denote by $S_n(D)$ the sequence number of a domain D stored at node n. Initially for every node n, $S_n(D)$ and $m_n(D)$ contain 0 and $-\infty$, respectively, for every domain D of which n is part or which is visible to n. The set $Border(D)$ contains the border nodes of D. A broadcast of a value m_D for domain D labeled with sequence number S_D is done by sending $UPD(D, m_D, S_D)$ to all nodes in the superdomain of D.

Algorithm III (network, multiple changes)

- Upon detecting a change in $m_n(n)$ to m'_n, node n performs $\text{Accept}_n(n, m'_n, S_n(n)+1)$.
- Upon receipt of $UPD(D, m_D, S_D)$, node n performs the following actions:

```
if S_D > S_n(D)
then if n ∈ D
     then S_n(D) := S_D;
          Broadcast_n(D, m_D, S_D)
              /* broadcast new sequence number of D */
     else Accept_n(D, m_D, S_D)
              /* update the m-value and
                 sequence number of D */
     fi
fi,
```

where

```
Broadcast_n(D, m_D, S_D) =
  Let D ◁ F in
  {for all {n, f}, f ∈ F
   do send UPD(D, m_D, S_D) to f
   od
  }

Accept_n(D, m_D, S_D) =
  Let D ◁ F in
  {m_n(D) := m_D;
   S_n(D) := S_D;
   Broadcast_n(D, m_D, S_D);
   if m_n(F) ≠ max{m_n(D_i) | D_i ◁ F}
   then if n ∈ Border(F)
        then Accept_n(F, max{m_n(D_i) | D_i ◁ F}, S_n(F)+1)
                 /* accept new value for F, broadcast it and
                    check m-value of the superdomain of F */
        else Accept_n(F, max{m_n(D_i) | D_i ◁ F}, S_n(F))
                 /* accept new value for F and check m-value
                    of the superdomain of F */
        fi
   fi
  }
```

□

3.3.3 Correctness Proof

In this section we prove that Algorithm III is correct. This proof consists of two parts. First, the property that after the algorithm has terminated (i.e., no node is involved in a broadcast) all nodes have correct values is shown to hold. Next, it is proven that the algorithm terminates within finite time, after the last change in a node's input has occurred. Before proving correctness, some notation is introduced. Node n *accepts* $UPD(D, m_D, S_D)$ if as a consequence of the receipt of this message, $\text{Accept}_n(D, m_D, S_D)$ is executed. The algorithm *has terminated for domain* D if no update message containing a value of D exists in the network and the algorithm has terminated for every subdomain of D.

Lemma 3.1 *Suppose that the algorithm has terminated for domain D and for all nodes $d, d' \in D$: $m_d(C) = m_{d'}(C)$, for all $C \lhd D$. Then the following holds for nodes $n, n' \in F$, with $D \lhd F$:*

- $m_n(D) = \max\{m_n(C) \mid C \lhd D\}$, *if* $n \in D$.
- $m_n(D) = m_{n'}(D)$.
- $S_n(D) = S_{n'}(D)$.

Proof: From the algorithm it directly follows that $m_n(D) = \max\{m_n(C) \mid C \lhd D\}$, if $n \in D$. Therefore, especially after termination this equality holds. It now follows from the fact that all nodes in D hold the same values for the subdomains of D, that if $n, n' \in D$: $m_n(D) = \max\{m_n(C) \mid C \lhd D\} = \max\{m_{n'}(C) \mid C \lhd D\} = m_{n'}(D)$.

Let m_D be this final value all nodes in D have for D and $S_D = \max\{S_n(D) \mid n \in D\}$. All nodes increment their sequence number if there is a neighbour with a higher sequence number for the same domain. Only border nodes are able to increment their sequence number for D to start a broadcast. Therefore, $S_D = \max\{S_n(D) \mid n \in \text{Border}(D)\}$. Every (border) node, which started a broadcast labeled with S_D, must have sent $UPD(D, m_D, S_D)$ to neighbour-nodes within the superdomain F. If this is not the case, a node n received the value m_D later and started a broadcast labeled with $S_n(D) > S_D$, which is in contradiction with the assumption that S_D is the maximum sequence number for D.

Every node n receiving $UPD(D, m_D, S_D)$, accepts it unless already $S_n(D) = S_D$ (and $m_n(D) = m_D$, if $n \notin D$). From the connectiveness of every domain it easily follows that every node $n \in F$, receives at least once $UPD(D, m_D, S_D)$.

□

Lemma 3.2 *Suppose the algorithm has terminated for all domains $C \lhd D$. Then within finite time the algorithm has terminated for domain D.*

Proof: Let $S_D = \max\{S_n(D) \mid n \in \text{Border}(D)\}$ at the moment the algorithm has terminated for all $C \lhd D$. After the algorithm has terminated for domains $C \lhd D$, no node $n \in \text{Border}(D)$ will start a new broadcast for domain D. The last broadcast for domain D (labeled with highest sequence number) must be labeled with S_D. As before, from connectiveness of every domain, it follows that every node $n \in F$, must receive $UPD(D, m_D, S_D)$ at least once. After a node accepts $UPD(D, m_D, S_D)$, it will not accept any update message labeled with a number less than or equal to S_D. Therefore, all earlier broadcast messages will be "absorbed" within finite time (because of the assumption that messages are not on a link for an infinite amount of time).

□

Theorem 3.1 *Within finite time after the last $m_n(n)$ has changed:*

- *the algorithm terminates for the whole network and*
- *the nodes contain correct values for all domains they are supposed to have a value for.*

Proof: We prove by induction on the structure of the hierarchical division that within finite time the algorithm terminates for every domain D and every node in the superdomain of D has correct values for D.

Base: D is a node. Because D does not have any subdomain, this case directly follows from Lemmata 3.1 and 3.2.

Step: Suppose D is not a node. From the induction hypothesis, within finite time the algorithm terminates for every $C \lhd D$ and for all nodes $d, d' \in D : m_d(C) = m_{d'}(C)$. From Lemmata 3.1 and 3.2 the step follows.

□

4 Formal Description

In this Section we formalize the steps as given in Section 3. The theory UNITY, a computational model and a proof system, is used. We assume the reader to be familiar with the notation used for the programs and the logic for the specification, design, and verification of UNITY programs. This theory can be found in chapters 2 and 3 of [1].

4.1 Problem specification

Let G be a graph in which an *input* is associated with each node. Inputs of nodes may change spontaneously. It is required to compute and maintain the maximum of all the inputs. Here, we describe and derive in UNITY notation the algorithms as described in Section 3.

4.1.1 Notation

- The set $node(G)$ contains the nodes of the graph G.
- $\forall n \in node(G) : m(n)$ contains the input of node n.
- Variable $m(G)$ should contain the maximum of the inputs.
- Let $[a_1, a_2, \ldots, a_n]$ denote the *bag*, or *multiset*, of elements $a_1, a_2, \ldots, a_n$.
  If $n_1, \ldots, n_x$ are the nodes of G, then $nodesval(G) = [m(n_1), m(n_2), \ldots, m(n_x)]$.
- $\vec{M}$ denotes a bag of integers.
- When the algorithm has terminated with value v we say that $term(G, v)$ holds. Then, the variable $m(G)$ equals the maximum v of all inputs, i.e.,

$$term(G, v) \quad \Rightarrow \quad v = m(G) = \langle \max n : n \in node(G) :: m(n) \rangle.$$

4.1.2 Specification

The specification of maintaining and obtaining the maximum value of the inputs in a network G, where each node has an input, can be formulated as follows:

SP1: $term(G, v) \wedge nodesval(G) = \vec{M}$ unless $nodesval(G) \neq \vec{M}$

SP2: $nodesval(G) = \vec{M} \mapsto \exists v : term(G, v) \vee nodesval(G) \neq \vec{M}$

4.2 Maximum computation on a division tree

Upon the graph G a hierarchical structure is imposed, which can be modeled as a division tree. Let the inputs of the graph G be placed as values at the leaves of this tree. By computing (i.e. obtaining and maintaining) at every internal node a value which is the maximum of the values at its sons, the root node of the tree obtains and maintains the maximum value of the inputs in G.

4.2.1 Notation

For every node D (of the tree):

- The set $leaf(D)$ contains D itself if D is a leaf or else the leaves under the node D.
- $m(D)$ is D's view of the maximum of the inputs under D.
- $m_D(C)$ is D's view of $m(C)$ for $C \lhd D$.
- Let $++$ be the *bag-union*. Then $leavesval(D)$ is defined as follows:
  $leavesval(D) = [m(D)]$, if $D \in leaf(D)$.
  Otherwise, let $C_1, \ldots, C_x$ be the sons of D, then
  $leavesval(D) = leavesval(C_1) ++ leavesval(C_2) ++ \ldots ++ leavesval(C_x)$.
  Note: $leavesval(R) \equiv nodesval(G)$, if R is the root of the division tree corresponding with the hierarchical division on G.
- The value of a node D must be made known to its father. $fatherknows(D, v)$ is true if the father of D has a correct view of $m(D)$.

$$fatherknows(D, v) \equiv \langle \wedge F : D \lhd F :: m_F(D) = v \rangle.$$

  If R is the root of the division tree, we define $fatherknows(R, v) \equiv true$.
- The algorithm is *finished* in some internal node D if the variable $m(D)$ contains a value v that is the maximum of all the values of its sons, and the algorithm has terminated (which is defined next) in these sons.

$$hfinished(D, v) \equiv \begin{cases} v = m(D) & \text{if } D \in leaf(D) \\ v = m(D) = \langle \max C : C \lhd D :: m_D(C) \rangle \wedge & \\ \qquad \forall C \lhd D : hterm(C, m(C)) & \text{otherwise.} \end{cases}$$

- The algorithm has *terminated* in some internal node D if the algorithm is finished in D and the value of D is known by the father of D.

$$hterm(D, v) \equiv hfinished(D, v) \wedge fatherknows(D, v).$$

  Note: $hterm(R, v) \equiv hfinished(R, v)$, if R is the root of the division tree.

4.2.2 The Solution Strategy: Formal Description

The specification of obtaining and maintaining the maximum value of the inputs at the leaves under a node D in a division tree, can be formally formulated as follows:

SP3: $hfinished(D, v) \wedge leavesval(D) = \vec{M}$ unless $leavesval(D) \neq \vec{M}$

SP4: $hterm(D, v)$ unless $\neg hfinished(D, v)$

SP5: $\forall C \lhd D : hterm(C, v_c) \mapsto (\exists v : hfinished(D, v)) \vee (\exists C : \neg hfinished(C, v_c))$

SP6: $hfinished(D, v) \mapsto hterm(D, v) \vee \neg hfinished(D, v)$

4.2.3 Proof of Correctness of the Solution Strategy

We prove that the unless-relation (SP1) and the progress-condition (SP2), given in the problem specification (in Section 4.2.1), are met by any strategy that satisfies conditions SP3, SP4, SP5 and SP6.

The proof that SP1 is met, follows directly from Lemma 4.1 which is presented next.

Lemma 4.1 $hterm(D,v) \equiv v = m(D) = \langle \max n : n \in leaf(D) :: m(n) \rangle \wedge fatherknows(D,v) \wedge \forall C \lhd D : hterm(C, m(C))$.

Proof: To prove the lemma, we use induction on the structure of the division-tree.

Base: D is a leaf. From the definition of $hterm(D,v)$, with D a leaf of the division-tree, this case directly follows.

Step: Suppose D is not a leaf and the lemma holds for all sons of D, i.e., $\forall C \lhd D : [hterm(C, v_c) \equiv v_c = m(C) = \langle \max n : n \in leaf(C) :: m(n) \rangle \wedge fatherknows(C, v_c) \wedge \forall B \lhd C : hterm(B, m(B))]$.

Then the following holds:

$$
\begin{aligned}
& hterm(D,v) \\
& \quad \equiv hfinished(D,v) \wedge fatherknows(D,v) \\
& \quad \equiv v = m(D) = \langle \max C : C \lhd D :: m_D(C) \rangle \wedge \forall C \lhd D : hterm(C, m(C)) \wedge \\
& \qquad fatherknows(D,v) \\
& \quad \{hterm(C, m(C)) \Rightarrow (m_D(C) = m(C))\} \\
& \quad \equiv v = m(D) = \langle \max C : C \lhd D :: m(C) \rangle \wedge \forall C \lhd D : hterm(C, m(C)) \wedge \\
& \qquad fatherknows(D,v) \\
& \quad \{\text{Induction hypothesis}\} \\
& \quad \equiv v = m(D) = \langle \max C : C \lhd D :: \langle \max n : n \in leaf(C) :: m(n) \rangle \rangle \wedge \\
& \qquad \forall C \lhd D : hterm(C, m(C)) \wedge fatherknows(D,v) \\
& \quad \equiv v = m(D) = \langle \max n : n \in leaf(D) :: m(n) \rangle \wedge \forall C \lhd D : hterm(C, m(C)) \wedge \\
& \qquad fatherknows(D,v).
\end{aligned}
$$

This proves the induction step.

We can conclude that $hterm(D,v) \equiv v = m(D) = \langle \max n : n \in leaf(D) :: m(n) \rangle \wedge \forall C \lhd D : hterm(C, m(C)) \wedge fatherknows(D,v)$.

□

Corollary 4.1 *If R is the root of the division tree corresponding to a hierarchical division on G, it is possible to define: $hterm(R,v) \equiv term(G,v)$.*

In the following we will use this definition. SP1 directly follows from this and SP3. We will now prove that SP2 is met.

Lemma 4.2

$$
\begin{aligned}
& (\forall C \lhd D : (hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c)) \mapsto \\
& \qquad (\forall C \lhd D : hterm(C, v_c)) \vee \neg(\forall C \lhd D : leavesval(C) = \vec{M}_c).
\end{aligned}
$$

Proof:

PSP theorem on SP3 and SP6:

$(hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c)) \mapsto$
$[(hterm(C, v_c) \vee \neg hfinished(C, v_c)) \wedge hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c]$
$\vee \neg(leavesval(C) = \vec{M}_c)$.

Rewriting the right side:

$(hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c)) \mapsto$
$[hterm(C, v_c) \wedge leavesval(C) = \vec{M}_c] \vee \neg(leavesval(C) = \vec{M}_c)$.

Rewriting the right side:

$(hfinished(C, v_c) \wedge leavesval(C) = \vec{M}_c)) \mapsto$
$hterm(C, v_c) \vee \neg(leavesval(C) = \vec{M}_c)$.

Conjunction over the $C's$ on the above:

$\forall C \lhd D : [(\mathit{hfinished}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c)) \mapsto$
$\mathit{hterm}(C, v_c) \vee \neg(\mathit{leavesval}(C) = \vec{M}_c)]$.

From SP4:

$\forall C \lhd D : [(\mathit{hterm}(C, v_c)$ unless $\neg(\mathit{hfinished}(C, v_C))]$.

The generalization theorem of the completion theorem on the above two:

$\forall C \lhd D : (\mathit{hfinished}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c) \mapsto$
$(\forall C \lhd D : \mathit{hterm}(C, v_C)) \vee \neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c \wedge \mathit{hfinished}(C, v_c))$.

From SP3, using simple conjunction:

$\forall C \lhd D : (\mathit{hfinished}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c)$ unless $\neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c)$.

The PSP theorem on the above two:

$\forall C \lhd D : (\mathit{hfinished}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c) \mapsto$
$[(\forall C \lhd D : \mathit{hterm}(C, v_C) \vee \neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c \wedge \mathit{hfinished}(C, v_c)))$
$\wedge(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c \wedge \mathit{hfinished}(C, v_c))]$
$\vee\neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c)$.

Rewriting the right side:

$\forall C \lhd D : (\mathit{hfinished}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c) \mapsto$
$\forall C \lhd D : [\mathit{hterm}(C, v_C) \wedge \mathit{leavesval}(C) = \vec{M}_c \wedge \mathit{hfinished}(C, v_c)]$
$\vee\neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c)$.

Weakening the right side:

$\forall C \lhd D : (\mathit{hfinished}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c) \mapsto$
$(\forall C \lhd D : \mathit{hterm}(C, v_C)) \vee \neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c)$.

□

Lemma 4.3

$$\forall C \lhd D : (\mathit{hterm}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c) \mapsto (\exists v : \mathit{hfinished}(D, v)) \vee \neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c).$$

Proof:

From SP3, using simple conjunction:

$\forall C \lhd D : (\mathit{hfinished}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c)$ unless $\neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c)$.

The PSP theorem on SP5 and the above:

$\forall C \lhd D : (\mathit{hterm}(C, v_c) \wedge \mathit{hfinished}(C, v_C) \wedge \mathit{leavesval}(C) = \vec{M}_c) \mapsto$
$[(\exists v : \mathit{hfinished}(D, v) \vee \neg(\forall C \lhd D : \mathit{hfinished}(C, v_C)))$
$\wedge\forall C \lhd D : (\mathit{hfinished}(C, v_C) \wedge \mathit{leavesval}(C) = \vec{M}_c)]$
$\vee\neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c)$.

Rewriting the left- and right side:

$\forall C \lhd D : (\mathit{hterm}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c) \mapsto$
$[\exists v : \mathit{hfinished}(D, v) \wedge \forall C \lhd D : (\mathit{hfinished}(C, v_C) \wedge \mathit{leavesval}(C) = \vec{M}_c)]$
$\vee\neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c)$.

Weakening the right side:

$\forall C \lhd D : (\mathit{hterm}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c) \mapsto$
$(\exists v : \mathit{hfinished}(D, v)) \vee \neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c)$.

□

Lemma 4.4 *For all domains D:*

$$(\mathit{leavesval}(D) = \vec{M}) \mapsto \exists v : \mathit{hfinished}(D, v) \vee \neg(\mathit{leavesval}(D) = \vec{M}).$$

Proof: We will give a prove by induction on the structure of the division-tree.
Base: If D is a leaf, this case trivially follows.
Step: Suppose D is an internal node.

From Induction Hypothesis:
$\forall C \lhd D : [(\mathit{leavesval}(C) = \vec{M}_c) \mapsto \exists v_c : \mathit{hfinished}(C, v_c) \vee \neg(\mathit{leavesval}(C) = \vec{M}_c)]$.
Rewriting the right side:
$\forall C \lhd D : [(\mathit{leavesval}(C) = \vec{M}_c) \mapsto$
$(\exists v_c : \mathit{hfinished}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c) \vee \neg(\mathit{leavesval}(C) = \vec{M}_c)]$.
From SP3:
$\forall C \lhd D : [(\mathit{hfinished}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c)$ unless $\neg(\mathit{leavesval}(C) = \vec{M}_c)]$.
Using the generalization of the completion theorem on the above two relations:
$(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c) \mapsto$
$[\forall C \lhd D : (\exists v_c : \mathit{hfinished}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c)$
$\vee \neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c)]$.
The cancellation theorem on the result of Lemma 4.2 and the above:
$(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c) \mapsto$
$[\forall C \lhd D : (\exists v_c : \mathit{hterm}(C, v_c)) \vee \neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c)]$.
Rewriting the right side:
$(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c) \mapsto$
$\forall C \lhd D : (\exists v_c : \mathit{hterm}(C, v_c) \wedge \mathit{leavesval}(C) = \vec{M}_c)$
$\vee \neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c)$.
Using transitivity on the result of Lemma 4.3 and the above:
$(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c) \mapsto$
$(\exists v : \mathit{hfinished}(D, v)) \vee \neg(\forall C \lhd D : \mathit{leavesval}(C) = \vec{M}_c)$.
Rewriting the left- and right side concludes the induction step, thereby proving the lemma.

□

Theorem 4.1 *SP2 is met by SP3, SP4, SP5 and SP6.*

Proof: Let R be the root of the division tree, then from Lemma 4.4 and corollary 4.1:

$$(\mathit{leavesval}(R) = \vec{M}) \mapsto \exists v : \mathit{hterm}(R, v) \vee \neg(\mathit{leavesval}(R) = \vec{M}).$$

Because $(\mathit{leavesval}(R) \equiv \mathit{nodesval}(G))$ and $(\mathit{hterm}(R, v) \equiv \mathit{term}(G, v))$, SP2 follows.
□

4.2.4 Derivation of a Program from the Specification

A program follows directly from the specification of the solution strategy. The function $inp(n)$ delivers initially the input of some leaf n. The possibility of the alteration of an input $m(n)$, of some leaf n, is implemented by the assignment $m(n) := g(n)$.

Program {maximum computation on a division tree}
initially
$\langle \mid D ::$
$m(D) = inp(D)$ if $D \in leaf(D) \sim -\infty$ if $\neg(D \in leaf(D))$
$\mid$
$\langle \mid C : C \lhd D :: m_D(C) = -\infty \rangle$
$\rangle$

assign
$\langle \| D ::$
$m(D) := g(D)$ if $D \in leaf(D) \sim$
$\langle \max C : C \lhd D :: m_D(C) \rangle$ if $\neg(D \in leaf(D))$
$\|$
$\langle \| C : C \lhd D :: m_D(C) := m(C) \rangle$
$\rangle$
end {maximum computation on a division tree}

4.3 Maximum computation on a divided graph

We now derive the distributed algorithm on a hierarchically divided network.

If D is an internal node of the division tree corresponding with the hierarchical division on the network, D corresponds with a domain in this network, and vice versa. Therefore in the following D may denote a node in the division tree or a domain in the hierarchically divided network. From the context it will always be clear what is meant.

4.3.1 Notation

For every domain D:

- The set $leaf(D)$ contains D itself if D is a node or else the nodes in domain D.
- $m_n(D)$ contains the value of D stored at node n.
- $S_n(D)$ contains the sequence-number of D stored at node n.
- Let $++$ be the *bag-union*. Then $Dleavesval(D)$ is defined as follows:
  $Dleavesval(D) = [m_D(D)]$, if $D \in leaf(D)$.
  Otherwise, let $C_1, \ldots, C_x$ be the subdomains of D, then
  $Dleavesval(D) = Dleavesval(C_1) ++ Dleavesval(C_2) ++ \ldots ++ Dleavesval(C_x)$.
  Note: $Dleavesval(D) \equiv leavesval(D)$
- $Dfatherknows(D, v)$ holds iff all nodes in the superdomain of D has v as value for D, i.e.,

$$Dfatherknows(D, v) \equiv \langle \wedge n : n \in F \wedge D \lhd F :: m_n(D) = v \rangle$$

- Let $n \in D$, n is ready for D iff the value v, as stored by n for D, equals the maximum of the values of the subdomains as kept by n.

$$Nfinished(n, D, v) \equiv \begin{cases} v = m_n(D) & \text{if } D \in leaf(D) \\ v = m_n(D) = \langle \max C : C \lhd D :: m_n(C) \rangle & \text{otherwise} \end{cases}$$

- $Dhfinished(D, v)$ holds iff all nodes are ready for D with value v and all subdomains have terminated (which is defined next), i.e.,

$$Dhfinished(D, v) \equiv \langle \wedge n : n \in leaf(D) :: Nfinished(n, D, v) \rangle \wedge \forall C \lhd D : (\exists v_c : Dhterm(C, v_c))$$

- $Dhterm(D, v)$ holds iff domain D has terminated with value v, that is all nodes in D are ready for D with value v and all nodes in the superdomain of D has v as value for D.

$$Dhterm(D, v) \equiv Dhfinished(D, v) \wedge Dfatherknows(D, v)$$

-

$$equal(F, D, \overline{S}) \equiv \langle \wedge n : n \in leaf(F) \wedge D \lhd F :: S_n(D) = \overline{S} \rangle$$

•

$$maxs(D,\overline{S}) \quad \equiv \quad \overline{S} = \langle \max n : D \lhd F \wedge n \in leaf(F) :: S_n(D) \rangle$$

4.3.2 The Solution Strategy: Formal Description

The specification of obtaining and maintaining the maximum value of the inputs at the nodes in a domain D by all the nodes in D, can be formally formulated as follows:

SP7: $Dhfinished(D,v) \wedge Dleavesval(D) = \vec{M}$ unless $Dleavesval(D) \neq \vec{M}$

SP8: $Dhterm(D,v)$ unless $\neg Dhfinished(D,v)$

SP9: $\forall C \lhd D : Dhterm(C,v_c) \mapsto (\exists v : Dhfinished(D,v)) \vee (\exists C \lhd D : \neg Dhfinished(C,v_c))$

SP10: $Dhfinished(D,v) \wedge maxs(D,\overline{S}) \mapsto equal(F,D,\overline{S}) \vee \neg Dhfinished(D,v)$

SP11: **Invariant**
$Dhfinished(D,v) \wedge equal(F,D,\overline{S}) \Rightarrow Dhterm(D,v)$

4.3.3 Proof of Correctness of the Solution Strategy

By defining,

$$(m_D(C) = v) \Leftrightarrow \langle \wedge n : n \in leaf(D) :: m_n(C) = v \rangle$$

the following equalities hold:

$$\begin{aligned} fatherknows(D,v) &\equiv Dfatherknows(D,v) \\ hfinished(D,v) &\equiv Dhfinished(D,v) \\ hterm(D,v) &\equiv Dhterm(D,v) \end{aligned}$$

From above SP3, SP4 and SP5 directly follow from SP7, SP8 and SP9 respectively.

We will now prove that SP6 is met by SP10 and SP11. Note that there is always a $\overline{S}$ such that $maxs(D,\overline{S})$ holds.

Suppose $maxs(D,\overline{S})$, then from the implication theorem:
$Dhfinished(D,v) \mapsto Dhfinished(D,v) \wedge maxs(D,\overline{S})$
Transitivity on the above and SP10:
$Dhfinished(D,v) \mapsto equal(F,D,\overline{S}) \vee \neg Dhfinished(D,v)$
Rewriting the right side:
$Dhfinished(D,v) \mapsto (equal(F,D,\overline{S}) \wedge Dhfinished(D,v)) \vee \neg Dhfinished(D,v)$
SP11:
$Dhfinished(D,v) \mapsto Dhterm(D,v) \vee \neg Dhfinished(D,v)$

From above and the observations made SP6 follows.

4.3.4 Derivation of a Program from the Specification

The set E contains all links existing in the network. A link between nodes n and b is denoted by $\{n,b\}$.

Program {maximum computation on a divided network using shared memory}
initially

$$\langle \|n,D,F : n \in F \wedge D \lhd F :: m_n(D), S_n(D) = \begin{array}{ll} inp(n), 1 & \text{if } n = D \sim \\ -\infty, \ 0 & \text{if } \neg(n = D) \end{array} \rangle$$

assign

$\langle \| n ::$

$\langle \| D, F : n \in D \wedge D \lhd F ::$

$m_n(D), S_n(D) := g(n), S_n(D) + 1 \quad$ if $n = D \sim$

$\langle \max C : C \lhd D :: m_n(C) \rangle, S_n(D) + 1$

if $((m_n(D) \neq \langle \max C : C \lhd D :: m_n(C) \rangle)$

$\wedge (n \neq D))$

$\| \langle \| b : b \in F \wedge \{n, b\} \in E :: S_n(D) := S_b(D) \quad$ if $(S_b(D) > S_n(D))$

$\rangle$

$\rangle$

$\|$

$\langle \| D, F : \neg(n \in D) \wedge (D \lhd F) \wedge (n \in F) ::$

$\langle \| b : b \in F \wedge \{n, b\} \in E :: m_n(D), S_n(D) := m_b(D), S_b(D) \quad$ if $(S_b(D) > S_n(D))$

$\rangle$

$\rangle$

$\rangle$

end {maximum computation on a divided network using shared memory }

The proof that the program satisfies the solution strategy is rather straightforward. SP7, SP8, SP9 and SP10 follow directly from the program text. SP11 follows from the following invariant, which can be easily proven from the program text. In the following variables i and j range over the set $\{1, \ldots, x\}$, with $x > 1$.

invariant

$[(n \in F) \wedge (n \notin D) \wedge (D \lhd F)] \Rightarrow$

$\langle \exists n_0, \ldots, n_x :$

$n_0 \in D \wedge \langle \wedge n_i n_j :: n_i \notin D \wedge n_i \in F \wedge ((i \neq j) \Rightarrow (n_i \neq n_j)) \rangle \wedge n_x = n ::$

$[(S_{n_i}(D) = S_{n_{i-1}}(D) \wedge m_{n_i}(D) = m_{n_{i-1}}(D)) \vee (S_{n_i}(D) < S_{n_{i-1}}(D))] \rangle$

This program can be implemented on a distributed system in the obvious way. The variables $m_n(D)$ and $S_n(D)$ are then local to n. A change in one of these variables is made known to the neighbours of n, which are in the superdomain of D, by sending it over the channels to these neighbours.

5 Conclusion

In this paper we derived a distributed algorithm on a hierarchically divided network. Every node in the network has an associated input-value and a variable in which it keeps the maximum of all inputs. These inputs may change spontaneously. Within finite time after a change, every node has updated its maximum value of the inputs.

The algorithm is developed in stages. After the problem description, an abstract algorithm on a (virtual) division tree is given. Next, the algorithm is transformed in the distributed algorithm that operates on a hierarchically divided network.

The stages are described informally as well as in UNITY. This combination appeared to be an ideal combination to design distributed and concurrent algorithms. It is our hope that the derivation and techniques used in the algorithm will be helpful in the future to design more distributed algorithms on hierarchically divided networks.

6 Acknowledgement

The authors thank Dr. G. Tel for his help during the course of this research.

References

[1] K. Mani Chandy and Jayadev Misra. *Parallel Program Design – A Foundation.* Addison-Wesley Publishing Company, Inc., 1988.

[2] Leslie Lamport. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming*, (2):175–206, 1982.

[3] P.J.A. Lentfert, A.H. Uittenbogaard, S.D. Swierstra, and G. Tel. Distributed hierarchical routing. Technical Report RUU-CS-89-5, Utrecht University, March 1989; also pp. 321-344 in *Proceedings Computing Science in the Netherlands CSN'89*, SION, Utrecht (1989).

[4] R. Perlman. Fault-tolerant broadcast of routing information. *Computer Networks*, 7:395–405, December 1983.

[5] Adrian Segall. Distributed network protocols. *IEEE Trans. Inf. Theory*, IT-29(1):23–35, January 1983.