Repeated Computation of Global Functions in a Distributed Environment

Vijay K. Garg and Joydeep Ghosh

Abstract—**In a distributed system, many algorithms need repeated computation of a global function. These algorithms generally use a static hierarchy for gathering necessary data from all processes. As a result, they are unfair to processes at higher levels of the hierarchy, who have to perform more work than processes at lower levels. In this paper, we present a new revolving hierarchical scheme, in which the position of a process in the hierarchy changes with time. This reorganization of hierarchy is achieved concurrently with its use. It results in algorithms that are not only fair to all processes, but also less expensive in terms of messages. The reduction in the number of messages is achieved by reusing messages for more than one computation of the global function. The technique is illustrated for distributed branch-and-bound problem, and for asynchronous computation of fixed points.**

Keywords— **Global Functions, Distributed Programs, Hierarchy, Permutations**

I. Introduction

In a distributed system, many algorithms compute a global function that requires information from all processes. These algorithms are sometimes called consensus protocols [13,2,3] , or total algorithms [19] Moreover, in many applications, the global function is computed several times [5]. Examples of applications which require repeated computation of a global function are deadlock detection [7] clock synchronization [11], distributed branch and bound [17], parallel alpha-and-beta search [9], global snapshot computation [6], and N+1-section search [1]. Examples of information necessary to compute the global function are local *wait-for* graphs for the deadlock detection problem, and the value of local bounds for distributed branch-and-bound search. Any centralized algorithm for gathering information is necessarily unfair towards the coordinator which has to do more work than others [15]. A centralized coordinator may also become a performance bottleneck. At the other extreme, an equitable ring-based algorithm takes a long time to collect the entire information [14, 19].

A common compromise is to logically map the processes onto a k-ary tree. Each process in the tree is responsible for relaying the information needed from its sub-tree to its parent. The root of the tree plays the role of coordinator. This approach guarantees that any process has to communicate with at most $k+1$ other processes. In addition, the intermediate processes may perform partial computations so that the root has less work to do. The approach is still unfair to processes at the higher levels of the tree who, in general, have to perform more work than processes at the lower levels.

This paper introduces a new revolving hierarchical scheme in which every process has to perform the same amount of work over time. In this scheme, the place of a process in the logical hierarchy changes with time. Moreover, information from previous hierarchies is used so that the reorganization of hierarchy is done concurrently with its use. This technique, when applied to any hierarchical algorithm, results in an algorithm that is not only fair to all processes, but also less expensive in terms of messages. The reduction in the number of messages is achieved by reuse of a message for more than one computation of the global function. We illustrate applications of this technique in distributed branch-and-bound problems and asynchronous computation of fixed points.

The idea of reorganization has appeared in the literature in many contexts. Many systems provide fault-tolerance by reorganizing the computation when a process/processor fails [21, 20, 16]. Worm programs [18] reorganize themselves to adapt to the availability of idle workstations and their failure. For example, a worm may consist of many more segments at night than during daytime. All the above systems adopt an ad-hoc approach to reorganization, which is done as an exception rather than a rule. Also, they emphasize fault-tolerance and not equitable workload distribution, which is the main aim of our scheme.

The algorithms in this paper are applicable to problems where the degree of each of the N processes in the underlying communication graph is at least $\Omega(\ log(N))$, and where the communication graph is known to all processes in the system. Similar conditions have been imposed for total algorithms [19], and consensus protocols [2,3] for computation of global functions. These approaches use the same algorithm several times if repeated computation of the global function is required, thus resulting in many wasteful messages. For example, K computations of a global function by [2] requires $O(K\ N\ log(N))$ messages. Our algorithms require only O(K N) messages.

This paper is organized as follows. Section II summarizes the desirable properties of a distributed data gathering problem. The properties that are desirable include light load on processes, high concurrency, and equitable workload distribution. We show that none of the existing methods satisfy all these properties. Section III describes the

This work was supported in part by the NSF Grant CCR 9110605, the Navy Grant N00039-91-C-0082, NSF Grant MIP-9011787, TATP Grant 14-9712, a TRW faculty assistantship award, and IBM Agreement 153.

Authors are with the Electrical and Computer Engineering Dept, University of Texas at Austin, Austin, TX 78712-1084; vijay@pine.ece.utexas.edu; ghosh@pine.ece.utexas.edu

revolving hierarchal scheme and shows that it possesses all the desirable properties outlined in Section II. The scheme is based on permutations which satisfy constraints that arise from the need to reuse messages and distribute the workload equally. A systematic method is given for generating such permutations. Section IV discusses an efficient implementation of the technique. Section V deals with a stricter requirement that no process at any step sends or receives more than one message, and presents a unifying scheme that can be used both for data gathering and results dissemination. Section VI generalizes the results of the previous sections for an arbitrary number of processes, and asynchronous communication. Section VII describes applications of our techniques.

II. Requirements for Distributed Data Gathering

In this paper, by a distributed system we mean a set of processes that communicate with each other using *synchronous* messages, that is, the sender of a message waits till the receiver is ready (as in CSP). This can be easily implemented by ensuring that the sender does not proceed till it receives an acknowledgement from the receiver. However, the latter part of the paper also discusses applications of our technique for distributed systems with asynchronous messages. It is assumed that transmission is error-free and none of the processors crash during the computation.

A distributed data gathering problem requires that one process receives enough data from everybody, directly or indirectly, to be able to compute a function of the global state. Let a time step of the algorithm be the time it takes for a process to send a message. Clearly, a process cannot send two messages in one time step. The desirable properties of any algorithm that achieves data gathering in a distributed system are:

1. Light Load: Let there be N processes in the system. No process should receive more than k messages in one time step of the algorithm, where k is a parameter dependent on the application, and on the physical characteristics of the network. A small value of k guarantees that no process is swamped by a large number of messages.

2. High Concurrency: Given the above constraint and the fact that there must be some communication, directly or indirectly, from every process to the coordinator process, it can be deduced that any algorithm takes at least $log_k(N)$ time steps. To see this, note that at the end of the first step, a process knows the state of at most $k+1$ processes. By the same argument, at the end of j^{th} time step, a process knows the state of at most $(k^j + k^{j-1} + k^{j-2}..+1)$ processes. It follows that at least $log_k(N) - 1$ step are required. The second requirement is that the algorithm must not take more than $O(log(N))$ steps.

3. Equal Load: For the purposes of load-balancing and fairness each process should send and receive the same number and the same size of messages over time. In addition, they should perform the same set of operations in the algorithm. This requirement assumes special importance for algorithms that run for a long time or when the processes belong to different individuals/organizations. The condition of equitable load is different from the symmetry requirement in [5,3], as processes in our algorithms can have different roles at a specific phase of the algorithm. However, in most practical applications, it is sufficient to ensure that all processes share the workload and responsibilities equally over time, rather than at every instant.

Let us consider the three main approaches taken for distributed data gathering, in light of the requirements stated above.

Centralized: In this scheme, every process sends its data directly to a pre-chosen coordinator. This scheme violates the requirements on light and equal load. The load on the coordinator can be reduced by constraining it to receive only k messages per time step, but then it takes N/k time steps to gather all the required information.

Ring-based: In this scheme, processes are organized in a ring fashion, and any process communicates directly only with its left and right neighbors. Ring-based algorithms can result in an equal load on all processes, but the level of concurrency is low since it takes $N-1$ time steps for one process to receive information from all other processes [8].

Hierarchy based: A logical k-ary tree is first mapped onto the set of processes. At every time step, each process sends states of processes in its sub-tree to its parent. This means that the root process receives information from all processes in $O(log(N))$ time. This approach also satisfies the constraint on the number of messages received per unit time; however, it violates the requirement of fairness, since processes at the higher levels of a hierarchy have to do more work than processes at the lower levels.

III. An Equitable, Revolving Hierarchy

In this section, we present an algorithm based on *revolving* hierarchy among processes [10], that satisfies all three desired properties of a distributed data gathering scheme. That is, the algorithm does not require a process to receive more than k messages per time step, computes the global function in $O(log(N))$ steps, and puts an equal work load on all processes.

Let there be N processes, numbered uniquely from the set $P = \{1, ..., N\}$, that are organized in the form of a k-ary tree. This tree also has N positions. Let $pos(x,t)$ be the position of the process x at time t. For simplicity, let $pos(x,0) = x$ for all $x \in P$. The reconfiguration of hierarchy consists of the remapping of processes to different positions. This reconfiguration is defined using a function $next : P \rightarrow P$ which gives the new position of the process which was earlier in position x. That is, if for some y and t, $pos(y,t) = x$, then $pos(y, t+1) = next(x)$. As two processes cannot be assigned the same position, $next$ is a 1-1 and onto function on the set P. Such functions are called permutations. Any permutation can be written as product of disjoint cycles [12]. For any permutation f defined on the set P, the orbit of any element $x \in P$ is defined to be:

$$orbit(x) = \{f^i(x) | i \geq 0\}$$

That is, $orbit(x)$ contains all elements in the cycle that contains x. f is called primitive if there exists a $x \in P$

time	message 1	message 2	idle
0	1,3 $\rightarrow$ 2	5,7 $\rightarrow$ 6	4
1	2,6 $\rightarrow$ 4	1,3 $\rightarrow$ 5	7
2	4,5 $\rightarrow$ 7	2,6 $\rightarrow$ 1	3
3	7,1 $\rightarrow$ 3	4,5 $\rightarrow$ 2	6

Fig. 1. A message sequence for repeated computation of a global function

such that $orbit(x) = P$. We require $next$ to be primitive so that any process occupies all positions in N time units.

As an illustration of a revolving hierarchy, consider the case when $N = 7$ and $k = 2$. Figure 1 shows a sequence of message transmissions that exhibit the desired properties outlined in Section II. At time $t = 1$, process 4 is able to obtain information from all other processes, since the messages received by it from processes 2 and 6 include the (possibly partially processed) messages sent by processes 1, 3, 5 and 7 in the previous time step. Thus it can compute a global function at the end of this time step. Similarly, at $t = 2$, process 7 can compute a global function.

The sequences of messages given in Fig. 1 is actually obtained by the revolving hierarchy illustrated in Fig. 2. To recognize this, consider an initial assignment of process i to node i of tree T_1, using an *inorder labeling*. At $t = 0$, the leaves of this tree send a message to their parents. At $t = 1$, we want to continue the propagation of these messages to the root of T_1, and simultaneously initiate messages needed for the next global computation. This can be achieved by defining another tree T_2 of N nodes such that the internal nodes of T_1 form one subtree of T_2, say the left subtree, and the leaf processes are remapped onto the root and the other subtree of T_2. The messages sent at $t = 1$ are precisely those sent by the leaf nodes of T_2 to their parents. Subsequent message sequences are obtained in a similar fashion by forming a new tree at each time step, as illustrated in Fig. 2. The trees $T_1, T_2, ...,$ are called *gather trees* since each such tree determines the sequence of messages used to collect all information required to compute one global function. Thus, a throughput of one global result per unit time is achieved after an initial startup delay of $\lceil \log N \rceil - 1$ steps. Note that this is possible because of the use of a message in $\lceil \log N \rceil - 1$ gather trees. Also, all messages may not be of equal size, since a message sent by a process may include a portion of the messages that it received in the previous time step. The actual content of messages is application dependent, and will be examined in Section VII. In this section, we shall concentrate on the sequence of messages generated, and on the properties that they satisfy.

The sequence of logical trees $T_1, T_2, ...,$ represents the time evolution of the assignment of the N processes to positions in a revolving tree. At every step, the processes are remapped onto the nodes of this tree according to a permutation function, $next(x)$, applied to the current position x, $1 \leq x \leq N$. For the example in Fig. 2, with an inorder

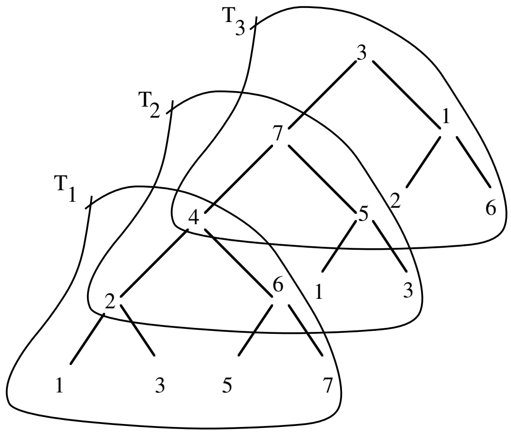

Fig. 2. Overlapping trees that determine message sequences

labeling of the nodes, this permutation is:

$$\begin{pmatrix} 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ 5\ 1\ 7\ 2\ 6\ 3\ 4 \end{pmatrix} \qquad (1)$$

Thus, process 1 which is in position 1 in T_1, goes to position 5 in T_2 and position 6 in T_3.

To generate a revolving hierarchy, $next(x)$ must satisfy the following two constraints:

1) *Gather Tree Constraint*: The interior nodes of T_i should form a subtree of T_{i+1}. That is, interior nodes at level j in T_i should be mapped to level $j+1$ in T_{i+1}, and the parent-child relationships among these nodes be preserved. This restriction ensures that the message sequences required for the root process at each snapshot to obtain global information are not disturbed during the reorganization needed to initiate messages for the next computation.

The following permutation function on inorder labels satisfies the gather tree constraint:

$$next(x) = x/2, \text{ for } even(x)$$

2) *Fairness Constraint*: The permutation should be primitive. This ensures that a process visits each position in the logical tree exactly once in N steps. Thus, if different positions require different workload, then each process will end up doing an equal amount of work after N time units.

We now present a permutation that satisfies gather-tree and fairness constraints. Define $lead0(x)$ as a function that returns the number of leading zeros in the n bit binary representation of x. For $x = 1, 2, ..., N = 2^n - 1$, consider the following $next(x)$ function:

```
next(x)
{
/* Type I move */
if (even(x)) then
```

$x' := x/2;$

/* Type II move */
if $(odd(x) \wedge (x < 2^{n-1}))$ then
$x' := x * 2^{lead0(x)} + 1;$

/* Type III move */
if $(odd(x) \wedge (x > 2^{n-1}))$ then
$x' := (x+1)$;
if $(x' = N+1)$ then $x' := x'/2;$

return(x');
}

The *next* function is applied to determine the next position of a process in an inorder labelled complete binary tree. Let the N nodes be divided into four disjoint groups:

Name	Members
$RInt$	$even(x) \wedge (x \geq 2^{n-1})$
$LInt$	$even(x) \wedge (x < 2^{n-1})$
$LLeaf$	$odd(x) \wedge (x < 2^{n-1})$
$RLeaf$	$odd(x) \wedge (x > 2^{n-1})$

Type I moves are required by the gather-tree constraint. Thus, if x is even it moves down the tree till it becomes a left leaf. Type II and Type III moves just visit the right subtree using inorder traversal. For a Type II move, $x * 2^{lead0(x)}$ gives the last node visited in the right subtree. The next node to be visited is obtained by adding 1 to the previous node visited. Note that as $x \in LLeaf$ for a Type II move, $lead0(x) \geq 1$, hence x' is odd. Also the *msb* of x' is 1, because x is multiplied by $2^{lead0(x)}$. Thus, a Type II move maps a left leaf node to a right leaf node. A Type III move just visits the next node in the inorder traversal, unless $x = N$ in which case x' is made to be the root to start the cycle all over again.

To show that *next* satisfies fairness and gather-tree constraints, we need a few Lemmas.

Lemma 1 *Let $f : P \to P$ be a permutation. Let $P_0, P_1, ..., P_{m-1}$ be a partition of P into m disjoint sets such that*

$$f(P_i) = P_{(i+1)\ mod\ m} \tag{2}$$

Then, f is primitive if and only if $\exists x \in P_0 : P_0 \subseteq orbit(x)$

Proof: If f is primitive, $orbit(x) = P$, and therefore includes P_0. We now show the converse. For any $x \in P_0$, $P_0 \subseteq orbit(x)$ implies that $\forall j : f^j(P_0) \subseteq f^j(orbit(x))$. Since $f(orbit(x)) \subseteq orbit(x)$, we get that $\forall j : f^j(P_0) \subseteq orbit(x)$. Further, as $f(P_i) = P_{(i+1)\ mod\ m}$, it follows that $\forall j : P_j \subseteq orbit(x)$. Hence, $P \subseteq orbit(x)$.

■

We say that $Q \subseteq P$ is a *core* of P with respect to f iff for any x that is in P, but not in Q, there exists an i such that $f^i(x) \in Q$. Intuitively, Q is any subset of P which has non-empty intersection with all cycles in P. We define restriction of a permutation $f : P \to P$ to its core $Q \subseteq P$ (denoted by $f_Q : Q \to Q$) as follows:
$f_Q(x) = f^j(x)$ where $j = min_{i \geq 1}\{i | f^i(x) \in Q\}$.

The following Lemma proves that f_Q is also a permutation.

Lemma 2 *If $f : P \to P$ is a permutation, then $f_Q : Q \to Q$ is also a permutation for any core Q of P with respect to f.*

Proof: We have to show that f_Q is a 1-1 and onto function. As both the domain and the range of f_Q are finite and have the same cardinality, it is sufficient to show that f_Q is 1-1. We show this by contradiction. Let $x, y \in Q$ such that $x \neq y$, but $f_Q(x) = f_Q(y)$. Let $k = min_{i \geq 1}\{i | f^i(x) \in Q\}$, and $l = min_{i \geq 1}\{i | f^i(y) \in Q\}$. k and l exist as Q is a core. Assume without loss of generality that $k \geq l$. Then, by definition of f_Q, $f^k(x) = f^l(y)$. As f is a permutation and therefore invertible, we deduce that $f^{k-l}(x) = y$. If $k = l$, we get that $x = y$, which is a contradiction. If $k > l$, we have found a strictly smaller number than k such that $f^{k-l}(x) \in Q$, again a contradiction.

■

The next Lemma provides the motivation of defining f_Q.

Lemma 3 *A permutation $f : P \to P$ is primitive iff there exists a core $Q \subseteq P$ such that f_Q is primitive.*

Proof: One side is obvious. If f is primitive, f_P is also primitive trivially. We show the converse. Let the permutation f not be primitive. This implies that f has a cycle C of length strictly smaller than $|P|$. Since Q is a core, there is no cycle in $P - Q$. This implies that C contains some but not all elements of Q, i.e., $C \cap Q$ is a non-empty proper subset of Q. Consider any $x \in C \cap Q$. Its orbit with respect to f_Q is also $C \cap Q$. Hence, f_Q also has a cycle smaller than $|Q|$, proving that f_Q is also not primitive.

■

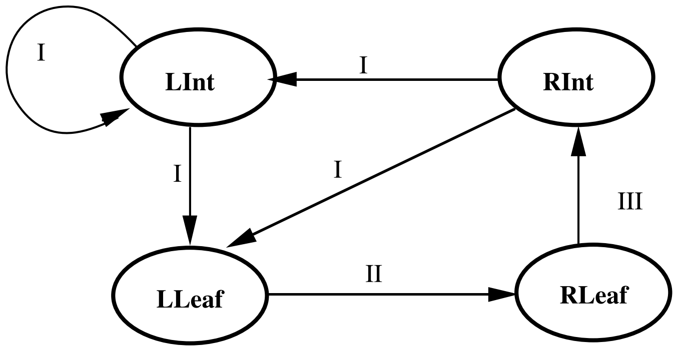

Fig. 3. Node groups and transitions

We are now ready for our first main result.

Theorem 1 *The function* next(.) *is a primitive permutation that satisfies the gather tree constraint.*

Proof: We first show that $next$ is a permutation. Let $x, y \in \{1, ..., N\}$ be such that $x \neq y$. Type I move is 1-1 because for any even x_1, x_2, $(x_1/2 = x_2/2)$ implies that $(x_1 = x_2)$. Type II move is 1-1, because for any odd x_1, x_2, if $lead0(x_1) \neq lead0(x_2)$, then $x_1 * 2^{lead0(x_1)} \neq x_2 * 2^{lead0(x_2)}$ as they have different number of trailing zeros. Otherwise, $x_1' = x_2'$ clearly implies that $x_1 = x_2$. Type III is also 1-1. Also, no element other than N maps to $(N+1)/2$ since the only other possibility, $x = (N+1)/2 - 1 = 2^{n-1} - 1$, does not belong to the domain of type III moves. Thus, if the same type of move is applicable for both x and y, then $next(x) \neq next(y)$ because each type of move (type I, type II, and type III) is 1-1. Furthermore, the ranges of different types of move are disjoint; for illustration see Figure 3. Hence, if different types of moves are applied to x and y, then also $next(x) \neq next(y)$. Therefore, $next$ is 1-1. Further, the domain and the range of $next$ have finite and equal cardinality, therefore it is also onto. Thus, it is a permutation.

To show that the permutation $next$ is primitive, first observe that $Q = LLeaf \cup RLeaf \cup RInt$ forms a core of P with respect to $next$. This is because for any $x \in LInt$, there exists i such that $next^i(x) \in LLeaf$. By Lemma 2, $next_Q$ is also a permutation. We now apply Lemma 1 to show that $next_Q$ is primitive. We partition Q into three sets $Q_0 = LLeaf, Q_1 = RLeaf$, and $Q_2 = RInt$. It can be easily checked that $next_Q(Q_i) = Q_{i+1 \ mod \ 3}$. Moreover, any cycle starting from a node x in RLeaf first visits vertex $x+1$ (or $(x+1)/2$) in $RInt$, followed by a vertex in LLeaf, which is followed again by the next vertex in Rleaf. Thus, the vertices in $RLeaf$ are visited in sequence, and $orbit(x) = RLeaf$. Applying Lemma 1, we conclude that $next_Q$ is primitive. As Q is a core of P and $next_Q$ is primitive, by applying Lemma 3, $next$ is also primitive.

Lastly, $next$ also satisfies the gather tree constraint because of Type I moves.

■

Significance: If $next(x)$ is used to determine the remapping of the processes to nodes for the next time step. in each time step, then:
(i) A global function can be computed in $\lceil \log N \rceil - 1$ steps after its initiation; and
(ii) A throughput of one global function computation per time step can be obtained.

Note that the gather trees are only tools to determine the sequence of message transmissions. The goal is to find at any time t, whether a given process needs to send a message, and if so, which process should be the recepient of that message.

Let $parent(x)$ yield the parent of node x, and $msg(x, t)$ be the process number to which process x sends a message at time t. If x does not send a message at time t, then $msg(x, t) = nil$. For an inorder labeling, a node has an odd label *iff* it is a leaf node. Since only leaf nodes send messages, we obtain:

$$msg(x,t) = \begin{cases} next^{-t}\ (parent\ (next^t(x))) & \text{if } odd(next^t(x)) \\ nil & otherwise \end{cases}$$

For an inorder labeling, the parent of a leaf node has the same binary representation as that node excepting that the two least significant bits are 10. For example, node 1010 is the parent of nodes 1001 and 1011. Thus, the parent can be readily evaluated.

IV. Implementation Issues

We can simplify the computation of $next^t(x)$ and $next^{-t}(x)$ by renumbering the tree nodes in the sequence traversed by a process. This is shown in Fig. 4, where the tree nodes are relabeled 0 through N-1. The old (inorder) labelling is given in parenthesis. [1] Let the processes be numbered 0,...,N-1 also, and process i be mapped onto node i at $t = 0$. This relabeling causes the $next(.)$ and $parent(.)$ functions to be transformed into $new\_next(.)$ and $new\_parent(.)$ respectively. Moreover, $new\_next^t(x)$ is simply equal to $x+t$. Therefore,

$$msg(x,t) = \begin{cases} new\_parent(x+t) - t & \text{if } x+t \text{ is a leaf;} \\ nil & otherwise \end{cases} \quad (3)$$

For $N = 31$, we obtain:

$$\begin{array}{rcccccccc} leaf\ node, i: & 0 & 15 & 7 & 22 & 3 & 10 & 18 & 25 \\ new\_parent(i): & 30 & 30 & 14 & 14 & 6 & 6 & 21 & 21 \\ \\ leaf\ node, i: & 1 & 4 & 8 & 11 & 16 & 19 & 23 & 26 \\ new\_parent(i): & 2 & 2 & 9 & 9 & 17 & 17 & 24 & 24 \end{array} \quad (4)$$

We only need to store the new_parent function for the leaf nodes to determine whom to send a message at any time t. Thus, the destination can be calculated in constant time, by looking up a table of size O(N). Alternatively, one can generate the new_parent function and trade off storage for computation time.

Let us define a *communication distance set*, CDS, as:

$$CDS = \{i \mid i = new\_parent(j) - j;\ j \ \text{ a leaf node}\}. \quad (5)$$

Lemma 4 *Process x will send a message (at some time) to process y* iff $y - x \in CDS$.

Proof: $\Rightarrow$: $y - x \in CDS$ means that there exists a leaf node j_1 such that $y - x = new\_parent(j_1) - j_1$.
Let $t_1 = j_1 - x$. Then $y - x = new\_parent(x+t_1) - (x+t_1)$ or $y = new\_parent(x+t_1) - t_1$. Since $(x+t_1) = j_1$ is a leaf, from Eq. 3 we infer that x sends a message to y at time t_1.
$\Leftarrow$: Let x send a message to y at time t_2. From Eq. 3, we have
$y = new\_parent(x+t_2) - t_2$ and that $x + t_2$ is a leaf node.
Substituting $j_2 = x + t_2$, we get
$y = new\_parent(j_2) - (j_2 - x)$, or $y - x = new\_parent(j_2) - j_2 \in CDS$
since j_2 is a leaf node.

[1] It can be shown that, even though the function *next(.)* gets transformed by changing the labeling of the tree nodes, the derived function, $msg(x,t)$, is unique for a given *next(.)* function.

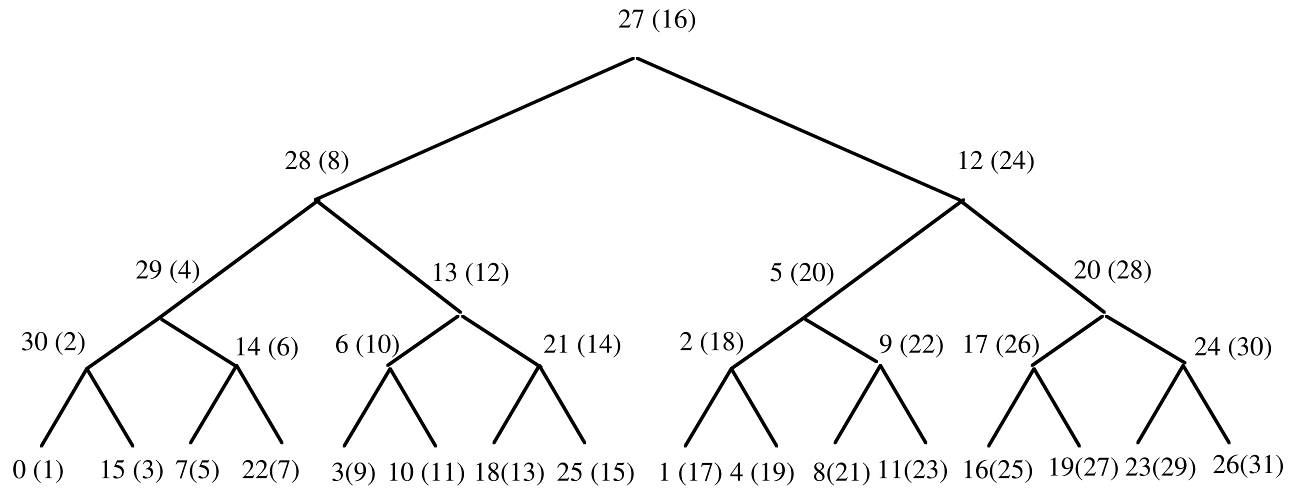

Fig. 4. Node labels generated by *next*. Original inorder labels are shown in parentheses.

∎

Using the above lemma one can define a communication graph corresponding to a given *next* function with a node for each process, and a directed edge (a, b) between two nodes only if a sends a message to b at some time. Each node of this graph has the same in-degree and out-degree, given by the size of the set CDS.

The *next* function is not the only permutation that satisfies the gather tree and fairness constraints. Type I moves are mandated by the gather tree constraint, but there are several choices for Type II and Type III moves. The following two criteria are proposed for chosing among several candidates for the *next* function:
a) If the derived *new_parent* function is simpler to generate, it is preferred.
b) A *next* function whose corresponding CDS set has a smaller size is preferred.

In the following, we show that the *next* function has CDS of size $2(log_2(N+1)-1)$.

We assume that the tree is labelled using inorder. Let $n = \log_2(N+1)$. We partition the set of 2^{n-2} left leaf nodes, $LLeaf$, into $n-1$ disjoint groups by defining
$LLeaf(i) = \{x \in LLeaf | lead0(x) = i\ \}$.
Note that since $b_{n-1} = 0$ and $b_0 = 1$, i takes values from 1 to $n-1$. The size of $LLeaf(i)$ is 2^{n-2-i} for $1 \leq i \leq n-2$, and 1 for $i = n-1$. The importance of this partition is that the cycle of permutation *next* visits a node in $LLeaf(i)$ after visiting exactly i internal nodes. This is because a right internal node is characterized by its most significant bit (msb) = 1, and each move of type I one adds one leading zero. All these moves except the last visit left internal nodes.

We partition the cycle of permutation *next* into 2^{n-2} segments. Each segment starts from a node in $RLeaf$ and ends in a LLeaf. The first segment starts at the leftmost leaf in $RLeaf$, which is labelled 1. Thus, we have partitioned all N elements into 2^{n-2} segments numbered from $1..2^{n-2}$.

Lemma 5 *The size of the segment* m *is* $trail0(m)+3$, *where* $trail0(m)$ *gives the number of trailing zeros in the binary representation of* m.

Proof: Nodes in RInt are visited in inorder by the definition of *next*. In an inorder traversal the height of i^{th} node visited is equal to the number of trailing zeros in binary representation of i. Thus, in segment m, we visit one node in $RLeaf$, one node at the height $trail0(m)$ in RInt, $trail(0)$ nodes in Lint, and one node in $LLeaf$ with the total of $trail0(m)+3$ nodes.

∎

Let $V(m)$ be the label of the left leaf node at the end of segment m. Clearly,

$$V(m) = \sum_{j=1}^{m} trail0(j) + 3m$$

Let $S(k) = \sum_{j=1}^{k} trail0(j)$. We need the following properties of $S(k)$.

Lemma 6 *1.* $S(a2^i) = a2^i - a + S(a)$ *for any* $i, a > 0$;
2. $S(2a.2^{i-1}) - S((2a-1).2^{i-1}) = 2^{i-1}$ *for any odd* a.

Proof: We use induction on i.
Base case:(i=1) We need to show that $S(2a) = a + S(a)$. We again use induction on a. It is true for $a = 1$ as $S(2*1) = 1 = S(1)+1$. Assume that it is true for $a < k$. Then $S(2k) = S(2k-2)+trail0(2k-1)+trail0(2k)$. Thus, using induction hypothesis,
$S(2k) = S(k-1)+(k-1)+trail0(2k-1)+trail0(2k)$.
Since $trail0(2k-1) = 0$ and $trail0(2k) = trail0(k)+1$, we get that
$S(2k) = S(k-1)+k-1+trail0(k)+1 = S(k)+k$.

Induction Assume that the Lemma is true for $i < k$.
$S(a2^k) = S(2a.2^{k-1})$. Using induction hypothesis, $S(a2^k) = 2a.2^{k-1} - 2a + S(2a)$. Using the base case to replace $S(2a)$, we get $S(a2^k) = a2^k - 2a + S(a) + a = a2^k - a + S(a)$

2. Using part 1, we get $S(2a.2^{i-1}) - S((2a-1).2^{i-1}) = 2^{i-1} - 1 + S(2a) - S(2a-1) = 2^{i-1} - 1 + trail0(2a) = 2^{i-1}$ as $trail0(2a)$ is 1 for any odd a.

∎

Lemma 7 *The nodes in $LLeaf(i)$ are labelled as $V((2a-1)2^{i-1})$, $a = 1, 2, 3, ..., 2^{n-2-i}$. Moreover, for an odd value of "a" (corresponding to a left child), the labels of the corresponding parent and right sibling are given by $V(a2^{i+1})-1$ and $V(a2^{i+1}+2^i)$ respectively.*

Proof: A segment m ends in $LLeaf(i)$ if and only if it visits exactly i internal nodes. From Lemma 5, the segment m visits exactly $trail0(m)+1$ internal nodes. Thus, segments ending in $LLeaf(i)$ are given by m such that $trail0(m)+1 = i$. Thus, m is of the form
$(2a-1)2^{i-1}$ for some $1 \leq a \leq 2^{n-2-i}$. We will now focus on those $LLeaf(i)$ which have more than 1 leaves, that is $1 \leq i \leq n-3$.

Then, odd values of a give the labels for left children, and even values for the right children in $LLeaf$. Since the nodes in $RInt$ at any level are visited from left to right,
(i) the parent of a left child in $LLeaf(i)$ is visited in the next segment that terminates in group $LLeaf(i+1)$. It terminates in group $LLeaf(i+1)$ because the parent of the child has same number of leading zeros as the child and the next element of the segment will have one more leading zero than the parent. The index of this segment is $(2a-1)2^{i-1} + 2^{i-1} = a2^i$.
(ii) the right sibling is visited in the next segment that terminates in group $LLeaf(i)$. The index of this segment is $(2a+1)2^{i-1}$.

∎

Theorem 2 *For $N = 2^n - 1$, the CDS for the* next(x) *labelling is of size $2(n-1)$, and its members are given by*

$$CDS = \bigcup_{i=1 \ to \ n-1} \{2^i - 1, \ -2^{i-1}\}. \qquad (6)$$

Proof: From Lemma 7, the contributions to CDS come from differences in labels of parents and leaves. Considering the nodes in group $LLeaf(i)$, $1 \leq i \leq n-3$, which are left children of their parents we get :
$V(a2^i) - 1 - V((2a-1)2^{i-1})$
$= S(a2^i) + 3a2^i - 1 - S((2a-1)2^{i-1}) - 3(2a-1)2^{i-1}$
$=2^{i-1} - 1 + 3.2^{i-1}$ (using Lemma 6)
$=2^{i+1} - 1$.

Considering the nodes in group $LLeaf(i)$, $1 \leq i \leq n-3$, which are right children of their parents we get:
$V(a2^{i+1}) - 1 - V(a2^{i+1} + 2^i)$
where a takes only odd values. On simplifying as before, this expression is equal to -2^{i+1}.

$LLeaf(n-2)$ and $LLeaf(n-1)$ contribute -1 and $2^{n-1} - 1$ Finally, the nodes in $RLeaf$ add 1 and -2 to the set CDS. Therefore, the CDS for the *next(x)* labelling is given by Eq. 6.

∎

Note that the CDS given by Eq. 6 is incremental, so that the communication set for a smaller number of communicating processes is a subset of the CDS for a larger number of processes. Also, the positive elements of the CDS are one less (mod N) in magnitude from some negative element. This means that the communication requirements can be satisfied by a homogeneous topology of degree $2n-1$ using bidirectional links and a two step communication scheme. In this topology, each node is connected to nodes at a distance of $\pm 2^i$, $0 \leq i \leq n-1$, as indicated in Fig. 5. Messages destined for a node at distance $2^i - 1$ for some i are sent in two steps. This topology preserves the incremental property which is attractive when mapping the processes onto a multicomputer system.

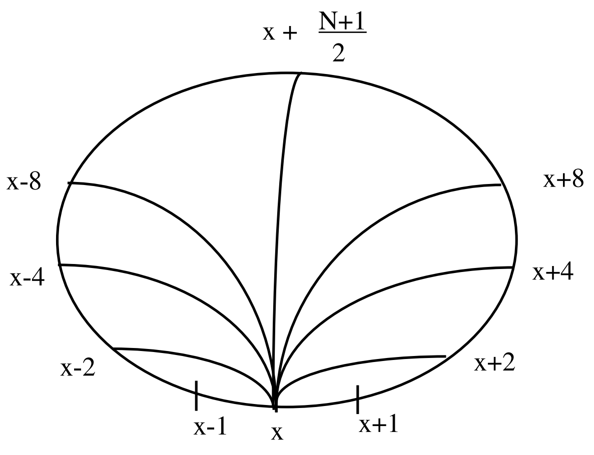

Fig. 5. Physical connectivity required based on a two-step routing procedure

V. Restricted Message Reception

In the previous sections, we proposed techniques for repeated computation of global functions where each process could receive messages from at most two other processes in a time slice. In this section, we consider a more restricted scenario in which a process can receive a message from only *one* other process in a given time slice, i.e., $k = 1$. Without loss of generality, let $N = 2^n$. A list representation is more convenient in this situation than the binary tree representation used in the previous section. Thus, if 5 sends a message to 1, and 2 to 8 in some time step, we can denote this by the list (5 1 2 8) or by the pairs, $5 \rightarrow 1$ and $2 \rightarrow 8$. The list positions are numbered as $0, 1, ..., 2^n - 1$.

Again, the message patterns in the next step can be determined by a suitable permutation, $snext(x)$. Let $b_{n-1}, ..., b_0$ be the binary representation of x, and $c_{n-1}, ..., c_0$ that of $x' = snext(x)$. Furthermore, let the operations $RS0$, $RS1$, $LS0$ and $LS1$ yield the numbers obtained by a right (left) shift of the bits with a 0/1 in the most (least) significant bit position.

The global function needs to be determined in $\log N$ steps, which is a tight lower bound for $k = 1$. If we draw an analogy with a knock-out tournament in which the receiving process is a winner, then the winners should play among themselves until there is a single winner. At the same time, the losers of the previous rounds also play to determine winners for following tournaments.

Thus, for the list representation, instead of the gather tree constraint, we have the following n *Tournament constraints*:

$b_0 = 1 \Rightarrow c_{n-1} = 0$; /* winners play among themselves */
for $i = 1$ *to* $n - 1$:
$(b_0 = 1) \wedge (b_{n-1}, ..., b_i = 0, ..., 0) \Rightarrow c_{n-1}, ..., c_{i-1} = 0, ..., 0$;
/* till the finals, yielding one winner. */

Consider the following function, where l is the number of consecutive zeros after the most significant bit, and $N = 2^n$:

```
  snext(x)
{
/* Type S1 move */
if  (b0 = 1) then x' := RS0(x);

/* Type S2 move */
if ( (b0 = 0) ∧ (b_{n-1} = 0)) then x' := 1, b_{n-2}, ..., b0 ;

/* Type S3 move */
if ((b0 = 0) ∧ (b_{n-1} = 1)) then
      if  (x' = N − 2) then x' := x + 1
      else x' := LS1^{l+1}(x) + 2;

return(x');
}
```

Figure 6 shows a partial sequence of the message patterns generated by $snext(.)$ with $n = 4$.

Theorem 3 *The function* snext(.) *satisfies both the fairness and the tournament constraints.*

Proof: The S1 moves guarantee that the tournament constraints are satisfied. Winning positions are characterized by $b_0 = 1$. In the next round, these positions are mapped onto the left half of the list so that the winners play among themselves. Moreover, this procedure is repeated recursively for each sublist of positions 0 through $2^i - 1$, $i = n-1$ down to 0, till we get a list of size two, denoting the "final" match.

To show the fairness constraint we divide the list positions into four equal sets: $ROdd$, $LOdd$, $LEven$ and $REven$, depending on the position being on the left ($b_{n-1} = 0$) or right half ($b_{n-1} = 1$) of the list, and whether the position is odd ($b_0 = 1$) or even. We observe that:
(i) S2 moves define a one-to-one mapping between $LEven$ and $REven$ positions;
(ii) S3 moves define a one-to-one mapping between $REven$ and $ROdd$ positions;
(iii) One or more consecutive invocations of S1 moves takes one from a position in $ROdd$ to a unique position in $LEven$
(iv) S3 moves ensure that the positions in $LEven$ are visited in sequence, i.e. the position $(x + 2)$ *mod* $(N/2)$ is visited after the position $x, x \in LEven$.
From Lemma 3 and arguments similar to Theorem 1, we get that snext(.) is a primitive permutation.

■

As in Section III, we can simplify the calculation of $snext(x)$ by relabelling the position numbers in the list in the sequence traversed by any process. For example, to obtain a function $n\_snext(x)$ from $snext(x)$ such that $n\_snext(x) = x+1 (\bmod N)$, the new labels for $N = 16$ are:

list position	0	1	2	3	4	5	6	7
label	4	3	8	2	11	7	14	1

(7)

list position	8	9	10	11	12	13	14	15
label	5	10	9	6	12	13	15	0

The new function, $n\_snext(.)$ is such that $n\_snext^t(x) = x + t$. If y is the new label of an even location in the list, then it sends a message to the label $dest(y)$ corresponding to the next odd position. For these positions, $rec(y) = nil$ signifying that no messages are received. If y is an odd location, then $dest(y) = nil$, signifying that no message is sent, while $rec(y)$ yields the label of the process from which it receives a message. For $N = 16$, we obtain:

y	0	1	2	3	4	5	6	7	8
$dest(y)$	*nil*	*nil*	*nil*	*nil*	3	10	*nil*	*nil*	2
$rec(y)$	15	14	8	4	*nil*	*nil*	9	11	*nil*

y	9	10	11	12	13	14	15
$dest(y)$	6	*nil*	7	13	*nil*	1	0
$rec(y)$	*nil*	5	*nil*	*nil*	12	*nil*	*nil*

(8)

At $t = 0$, let process x be in position labeled x in the list. Then, for $t \geq 0$,

$$msg(x, t) \quad = \quad dest(x + t) - t. \tag{9}$$

The communication distance set is:

$$CDS = \{i \mid i = dest(j) - j; \;\; dest(j) \neq nil\}. \tag{10}$$

time	messages (sender → receiver)							
0	$4 \rightarrow 3$	$8 \rightarrow 2$	$11 \rightarrow 7$	$14 \rightarrow 1$	$5 \rightarrow 10$	$9 \rightarrow 6$	$12 \rightarrow 13$	$15 \rightarrow 0$
1	$3 \rightarrow 2$	$7 \rightarrow 1$	$10 \rightarrow 6$	$13 \rightarrow 0$	$4 \rightarrow 9$	$8 \rightarrow 5$	$11 \rightarrow 12$	$14 \rightarrow 15$
2	$2 \rightarrow 1$	$6 \rightarrow 0$	$9 \rightarrow 5$	$12 \rightarrow 15$	$3 \rightarrow 8$	$7 \rightarrow 4$	$10 \rightarrow 11$	$13 \rightarrow 14$
3	$1 \rightarrow 0$	...						

Fig. 6. Message Sequences Generated by *snext*

For the $snext(.)$ function defined above, with N = 16, we get:

$$CDS = \{1, 3, 5, -6, -4, -3, -1\}.$$

As in Section IV, we would like to determine a lower bound for the size of CDS. The labelling of the list positions by *n_snext2(x)* described below, results in a CDS of size $\log N$. List position 0 is labelled as 0 by *n_snext2(x)* to form a convenient starting point. The position x' to be labelled next is determined from the current list position, x, as follows:

if $b_0 = 1$ **then** $x' :=$ RS0(x);
else if $(b_0 = 0) \wedge (b_{n-1} = 1)$ **then** $x' := x + 1$;
else $x' :=$ first available position in $REven$
(from left to right).

The labels generated by *n_snext2(x)* for $N = 16$ are:

$$\begin{array}{rcccccccc} \textit{list position} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \textit{label} & 0 & 15 & 7 & 14 & 3 & 6 & 10 & 13 \\ \textit{list position} & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \textit{label} & 1 & 2 & 4 & 5 & 8 & 9 & 11 & 12 \end{array} \quad (11)$$

The corresponding CDS is { 1, 3, 7, 15 }.

The labelling obtained by *n_snext2(x)* is similar to the *new_next2(x)* labelling given in Section IV. The labels of the $LEven$ positions are given by the numbers in $V_n(i)$, $0 \leq i \leq N/4 - 1$. We can group the positions in $LEven$ into sizes of N/8, N/16,...,2,1, with the ith group being characterized by $b_{n-1}, ..., b_{n-i-1} = 0, ..., 0, 1$, excepting for the last group which consists solely of position 0. The labelling can be analyzed as before through a sequence of segments, each starting at an $REven$ position, visiting the next $ROdd$ position and terminating at an $LEven$ position via none or more $LOdd$ positions. It can be seen that the $Leven$ positions in the ith group contribute the number $V(2^i) - V(2^{i-1}) - 1 = 2^{i+1} - 1$ to the CDS. Also the number 1 belongs to the CDS since the label of an $ROdd$ position is one more than the label of the preceeding $REven$ position. This yields the following result:

Theorem 4 *For $N = 2^n$, the CDS for the* n_snext2(x) *labelling is of size n, and its elements are given by*

$$CDS = \bigcup_{i=1 \ to \ n} \{2^i - 1\}. \quad (12)$$

A. Broadcasting of Messages

In several applications, such as the distributed branch-and-bound algorithm explained in Section VII, the result R of a global computation also needs to be transmitted to all the processes. In this section, we show that if $snext(x)$ satisfies some further conditions, then such broadcasts can be performed by attaching a copy of the result to the *same* set of message sequences that are used to gather information for future computations of R. Furthermore, this broadcast is achieved in $\log(N)$ time steps, which is the lower-bound for the single sender case.

To be able to broadcast in $n = \log N$ steps, the number of processes having a copy of R must double at each step. This means that each of these processes must become a sender of a message in the next time step, and the recipients of these messages must be processes that have not yet obtained a copy of R.

We first observe that the message sequence shown in Fig. 6 does not satisfy the broadcasting requirements. At $t = 0$, process 4 computes R. At $t = 1$, a copy of R is passed on to process 3. These two processes further pass on copies of R to 2 and 9 respectively in the next time step. However, at $t = 3$, we see that 4, which already has a copy of R, is a receiver again. Therefore, the number of processes to whom R is broadcast after 3 steps is less than 2^3. Clearly, $snext(.)$ needs to satisfy additional constraints to double as a broadcasting function.

Theorem 5 *Let $b_{n-1}, ..., b_0$ be the current position of a process, and $c_{n-1}, ..., c_0$ be its next position as indicated by $snext(.)$. The function* snext(.) *can also perform a broadcast of result R in n time steps provided the following additional $n-1$ constraints are met:*

$$b_i, ..., b_1 = 0, ..., 0 \Rightarrow c_{i-1}, ..., c_0 = 0, ..., 0; \text{ for } i = 1 \text{ to } n{-}1. \quad (13)$$

Proof: The process that computes R at time t_0 is in position 1 at that instant. We show by induction that, at time $t_0 + j$, $j = 1$ to n, the 2^j processes whose positions at time $t_0 + j$ are characterized by $b_{n-j}, ..., b_1 = 0, ..., 0$, have a copy of R. This assertion is clearly true for $j = 1$. Assume that it is valid for $j = m \leq n - 1$. The constraints given by Eq. 13 guarantee that, at the next time step, all the processes that already have a copy of R will be in a sending position, ($c_0 = 0$), characterized by $c_{n-m-1}, ..., c_0 = 0, ..., 0$. Furthermore, these positions will be unique since $snext(.)$ is a permutation. Each of these processes can convey a copy of R to the processes occupying positions $c_{n-m-1}, ..., c_1 = 0, ..., 0; c_0 = 1$. Thus, at time $t_0 + m + 1$, the 2^{m+1} processes in positions with $b_{n-m-1}, ..., b_1 = 0, ..., 0$ can obtain a copy of R.

■

On examining $snext(.)$, we see that it was not able to perform a concurrent broadcast because the S3 moves failed to satisfy Eq. 13. Now consider the partial sequence of messages shown in Figure 7. The reader can verify that a global function is broadcast in 4 steps after it is computed, if this sequence is used.

The message sequence of Fig. 7 was generated by the function *bcnext*(.) given below:

```
bcnext(x)
{
/* Type S1 move */
if   (b0 = 1) then x' := RS0(x);

/* Type S2 move */
if ( (b0 = 0) ∧ (b1 = 0)) then x' := RS1(x) ;

/* Type S3 move */
if ((b0 = 0) ∧ (b1 = 1)) then
      x' := LS1^a((LS0^b(x) + 2) mod 2^(n-1));

return(x');
}
```

where a and b are the number of leading zeros and ones respectively, in the argument.

The right-shifts cause the constraints of Eq. 13 to be automatically satisfied for S1 and S2 moves. For S3, $b_1 = 1$, so the constraints do not apply. Therefore, *bcnext*(.) satisfies the broadcast requirements. Moreover, it can be easily show that bcnext is a primitive permutation. Therefore, we have:

Theorem 6 *The function* bcnext(.) *satisfies the broadcasting, fairness and tournament constraints, and therefore generates message sequences that:*

1. *allow a new global computation at every time step* t, $t \geq \log N$;
2. *enable a process to gather information for a global computation in* $\log N$ *steps; and*
3. *enable broadcast of the results of a global computation to all processes in* $\log N$ *steps.*

VI. Extensions

This section shows that the technique to generate an admissible permutation for a binary tree can be generalized to any k-ary tree. The revolving hierarchy scheme is also shown to apply even when it is not possible to impose a *complete* k-ary tree on the network, and also when asynchronous messages are used instead of synchronous messages.

General k: We have shown the methods to generate suitable permutations for binary trees. The technique easily generalizes to any k-ary tree. A complete k-ary tree of height n has k^n leaves, which can be divided into k groups of equal size corresponding to the k subtrees rooted at the children of the root of the k-ary tree. The behavior of any suitable permutation, k-ary next function on internal nodes is unique due to the gather-tree constraint, and is similar to type I move of Theorem 1. The k-ary next function needs to define a 1-1 mapping from leaves in one group to leaves in the successive group using a move similar to type II in Theorem 1. Finally, the last leaf group is mapped to internal nodes using type III move.

General N: So far we had assumed that $N = (k^j - 1)/(k-1)$, so that a complete k-ary tree could be used. Given any general N, we can find j such that $k^{j-1} - 1 < (k-1)N \leq k^j - 1$. We now supplement the network with enough virtual nodes so that the total number of nodes can form a complete tree. Thus, the number of virtual nodes is

$v' = (k^j - 1)/(k-1) - N < (k^j - k^{j-1})/(k-1)$
$= k^{j-1} < N(k-1) + 1.$

This implies that if the load of virtual nodes is distributed fairly, no node has to carry the burden of more than $k-1$ virtual nodes. A real node sends and receives messages on behalf of the virtual nodes it is responsible for. We can reduce the maximum load on any node, by reducing the arity of the tree at the expense of increasing its height.

Asynchronous Messages: So far we had assumed that the communication is done via synchronous messages. To see that the technique works even with asynchronous messages, note that every process becomes root in any consecutive N steps. This process must receive messages directly, or indirectly from all processes. It relinquishes its position as the root only after receiving all information needed to compute a global function. This property automatically synchronizes the algorithm. Observe that algorithms for distributed search in Section VII work even if the messages are asynchronous.

VII. Applications

Our techniques can be applied to derive algorithms for a wide variety of distributed control problems, especially those requiring computation of *asynchronous* global functions. In an asynchronous global function, if information from a process is available regarding two different times, the older information can always be discarded. For example, consider a distributed implementation of a branch-and-bound algorithm for the minimum traveling salesman path(TSP) problem. Each processor explores only those partial paths which have cost smaller than the minimum of costs of all known complete paths. If a processor knows of a path with cost 75 at time step t and another of cost 70 at time step $t+1$, then it needs to propagate only 70 as the cost of its current minimum path. In this example, the root does not need the current best path determined by each processor at each time step to compute the (current) global minimum. The states that it receives may be staggered in time, i.e. its own state may be current whereas the state of its sons one phase old, and the state of its grandsons two phases old. We next describe our technique for two problems which satisfy the asynchrony condition on the global function. These are distributed branch-and-bound algorithms, and distributed computation of fixed points.

A. Distributed Branch-and-Bound Algorithms

These algorithms are most suitable for our technique. They satisfy not only the asynchrony condition, but also

time	messages (sender → receiver)							
0	$4 \rightarrow 3$	$10 \rightarrow 2$	$12 \rightarrow 9$	$15 \rightarrow 1$	$5 \rightarrow 11$	$13 \rightarrow 8$	$6 \rightarrow 14$	$7 \rightarrow 0$
1	$3 \rightarrow 2$	$9 \rightarrow 1$	$11 \rightarrow 8$	$14 \rightarrow 0$	$4 \rightarrow 10$	$12 \rightarrow 7$	$5 \rightarrow 13$	$6 \rightarrow 15$
2	$2 \rightarrow 1$	$8 \rightarrow 0$	$10 \rightarrow 7$	$13 \rightarrow 15$	$3 \rightarrow 9$	$11 \rightarrow 6$	$4 \rightarrow 12$	$5 \rightarrow 14$
3	$1 \rightarrow 0$	...						

Fig. 7. Message sequence generated by *bcnext*

have an additional attractive property: it is feasible for internal nodes to perform some intermediate operations and reduce the overall state sent to their parents. For example, in the TSP problem, an internal node needs to forward only that message which contains the minimum traveling path and not all the messages it received from its children. Thus, a hierarchical algorithm (static or dynamic) for this problem reduces the total amount of information flow within the network. In general, if the required global function is associative in its arguments (such as *min*), then information can be reduced by performing operations at internal nodes.

A distributed branch-and-bound problem requires multiple processors to cooperate in search of a minimum solution. Each processor reduces its search space by using the known bound on the required solution. In our description of the algorithm, we assume that *search (knownbound)* procedure searches for a solution for some number of steps and returns the value of its current minimum solution. The crucial problem then, is the computation of the global bound and its dissemination to all processes. To solve this problem, we apply the results obtained in Section V which permit us to use the same permutation for the gather tree and the broadcast tree. This permutation is implemented by means of *tosend* and *torec* functions as described earlier. The function *tosend* returns -1 if no message needs to be sent in the current time step. In the algorithm described below, we have assumed that at most one message can be received in one time step.

```
Process i;
var
  knownbound, mymin, hismin: real;
  step, numsteps, dest:integer;
begin
  Initialization:
  knownbound := infinity;
  for step:=0 to numsteps do
  begin
    mymin := search(knownbound);
    dest = tosend(i, step);
    if (dest <> -1) then
       send(dest, mymin)
    else begin
       receive(torec(i, step), hismin);
       knownbound := min(mymin, hismin);
    end;(* else *)
  end; (* for *)
end;(* process i *);
```

Each process uses *tosend* and *torec* to find out when and with whom it should communicate. From Theorem 6, each process receives a global minimum bound every $2.log(N)$ steps, and sends/receives an equal number of messages.

A static hierarchical algorithm for this problem requires $2(N-1)$ messages per computation of a global function: $N-1$ messages for the gather-tree, and $N-1$ messages for the broadcast tree. Each message is of constant size required to represent the minimal solution known to the sender. Our algorithm requires only $N/2$ messages, which is about four times less expensive than the static hierarchical algorithm. The reduction in the number of messages does not lead to any increase in the size of messages. It is obtained by reusing a message for multiple global function computations. Moreover, our algorithm exhibits a totally fair workload distribution - each process has to send and receive an equal number of messages.

B. Asynchronous Distributed Computation of Fixed Points

This problem exemplifies the class of *asynchronous* global functions which do not allow reduction of information at internal nodes. Assume that we are given N equations in N variables. We are required to find a solution of this set of equations. Formally, we have to determine x_i such that, $x_i = f_i(x_1, x_2, ..., x_N)$ for all $1 \leq i \leq N$.

This problem arises in many contexts, such as computation of stationary probability distributions for discrete Markov chains. Moreover, an iterative asynchronous computation of these equations will yield their solution under conditions posed in [4] We assume that equations are on different processors, and every processor computes one coordinate of the x vector. In the algorithm given below, we have used an array t to record the time step at which values of x coordinates are computed.

```
Process i;
var
  (* N is the number of processes *)
  x, hisx: array[1..N] of real;
  t, hist: array[1..N] of integer;
  (* t[j] = time step for which x[j] is known *)
  j,step: integer;
begin
  step := 0;
  x[i] := initial; t[i] := step;

  (* values of x[j] are not known at time 0 *)
  for j:=1 to N do
      if (j<>i)  x[j],t[j]:=0,-1;
```

```
  while (not fixed_point) do
  begin
     dest = tosend(i, step);
     if (dest <> -1) then
        send(dest, x,t)
     else begin
        receive(torec(i, step), hisx, hist);
        (* update coordinates of my vector *)
        for j:=1 to N do
          if hist[j] > t[j] then
              x[j], t[j] := hisx[j], hist[j];
        (* recompute my coordinate *)
        x[i] := f_i[x];
        t[i] := step;
     end;(* else *)
     step := step + 1;
  end;(* while *)
end;(* process i *);
```

Each process in the above algorithm sends or receives the x vector using *tosend* and *torec* primitives. On receiving an x vector, it updates the value of any coordinate $x[j]$ which has its $t[j]$ less than the received $hist[j]$. These steps are repeated till the computation reaches a fixed point. We have not considered the detection of fixed point in the above algorithm. To detect the fixed point, it is sufficient to note that if a process on becoming root finds that its x vector has not changed since the last time, then the computation must have reached its fixed point. To ensure that all processes terminate at the same step, any process that detects fixed point should broadcast a time step when all processes must stop. The details are left to the reader.

The algorithm requires N/2 messages per computation and broadcast of the global computation. The message size in this algorithm is of order O(N) assuming that it requires a constant number of bits to encode state of one process. This size can be reduced at the expense of time required for propagation of a change as follows. In the above algorithm, a change in any coordinate is propagated to all processes within $2log(N)$ steps. This is because any change in a process is gathered in $log(N)$ steps by a root process due to tournament constraints, and propagated to all other processes in another $log(N)$ steps due to broadcast constraints. We observe that even if broadcast constraints are not used, every process will receive the change in $O(N)$ steps due to fairness constraints. This property can be exploited to reduce the message size by requiring every process to send states of only a selected set of processes instead of the entire system. Let there be $N = 2^n$ processes in the system. At every time step, 2^j processes need to send states of only 2^{n-j-1} processes for values of j between 0 and $n-1$. That is, one process needs to send states of N/2 processes, two processes need to send states of N/4 processes, and so on. Therefore, the total number of bits sent in any time step is

$$\sum_{i=0}^{n} 2^i.(N/2^{i+1}) = O(nN) = O(N\ log(N))$$

Thus, on an average a message is of $O(log(N))$ size.

VII. Conclusions

We have presented a general technique for repeated computation of global functions in a distributed environment. Our technique is based on a new dynamic hierarchical scheme. This hierarchical scheme determines the messages that need to be sent at any given time. As the computations evolve, the hierarchy changes in such a way that it results in an equitable distribution of work among all processes.

Our techniques, when applied to a large class of distributed algorithms, not only result in an even workload, but also lower communication overheads by reducing the total number of messages. We have successfully applied these techniques to problems such as distributed branch-and-bound and distributed asynchronous fixed-point computation.

Some related issues still need to be resolved. First, the choice of a permutation, on which the message patterns generated depends, is not unique. Recollect that the logical neighbors for communication is given by the set CDS corresponding to the chosen permutation. An implementation issue is to keep this set small and easily mapable onto the physical interconnection network. A systematic scheme for including connectivity considerations in selecting a permutation remains an open problem.

We have assumed error-free transmission of messages in this paper. Generalization of our techniques in the presence of faulty communication channels or malicious processes is a topic of future research.

Acknowledgements

We would like to thank anonymous referees for their helpful suggestions on an earlier version of this paper.

References

[1] S.G.Akl, *The Design and Analysis of Parallel Algorithms*, Prentice Hall, 1989.

[2] J-C. Bermond, J-C. Konig, and M. Raynal, "General and Efficient Decentralized Consensus Protocols," *Distributed Algorithms, 2nd International Workshop*, Amsterdam 1987, Lecture Notes in Computer Science 312, Springer Verlag 1988, pp 41-56.

[3] J-C. Bermond, and J-C. Konig, "General and Efficient Decentralized Consensus Protocols II," *Parallel and Distributed Algorithms, International Workshop*, Paris 1988, North Holland 1989, pp 199-210.

[4] D.P.Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computation*, Prentice Hall, 1989.

[5] L. Bouge, "Repeated Snapshots in Distributed Systems with Synchronous Communication and their implementation in CSP," *Theoretical Computer Science*, Vol. 49, 1987, pp 145-169.

[6] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining global states of distributed systems", *ACM Transactions on Computer Systems* 3(1):63-75, February 1985.

[7] K.M. Chandy, J. Misra, and L. Haas, "Distributed Deadlock detection", *ACM Transactions on Computer Systems* 1(2):145-156, May 1983.

[8] E.W. Dijkstra, W.H.J. Feijen and A.J.M. VanGasteren, "Derivation of a Termination Detection Algorithm for Distributed Computation," *Information Processing Letters*, 16, June 1983, pp 217-219.

[9] R.A. Finkel, and J.P. Fishburn, "Parallelism in alpha-beta search," *Artificial Intelligence* 19,89-106.

[10] V.K. Garg and J. Ghosh, "Symmetry In Spite of Hierarchy," Proc. 10th IEEE International Conference on Distributed Computing Systems, Paris, France 1990, pp 4-11.

[11] R. Gussella, "Tempo: A clock synchronization algorithm", Tech. Report, Computer Science Division, University of California, Berkeley, 1986.

[12] I.N. Herstein, Topics in Algebra, Wiley Eastern Limited 1975.

[13] T.V. Lakshman and A.K. Agrawala, "Efficient Decentralized Consensus Protocols," IEEE Transactions on Software Engineering, vol. SE-12, no. 5, 1986, pp 600-607.

[14] G. Le Lann. "Distributed Systems - Toward a Formal Approach", *Proc. AFIP Congress 77*, 1977, pages 155-160.

[15] D. Menasce and R.R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases". *IEEE Transactions on Software Engineering*, SE-35, No. 3, May 1979, pp. 195-202.

[16] M.L.Powell and D.L.Presotto, "Publishing - A reliable broadcast communication mechanism", *Operating Systems Review*, 17, 5, 100-109.

[17] M.J.Quinn, *Designing Efficient Algorithms for Parallel Computers,* McGraw-Hill, 1986.

[18] J. F. Shoch and J. A. Hupp. "The "worm" programs–Early experience with a distributed computation " *Communications of the ACM*, 25(3):172-180, March 1982.

[19] G. Tel, "Total Algorithms," *Parallel and Distributed Algorithms, International Workshop*, Paris 1988, North Holland 1989, pp 187-198.

[20] W.T.Tsai, "The design and maintenance of large hierarchical networks", Ph.D. dissertation, University of California, Berkeley, 1985.

[21] A.M. Van Tilborg, and L.D.Wittie, "Wave Scheduling: Distributed allocation of task forces in network computers," *Proc. of the 2nd International Conference on Distributed Computing Systems*, IEEE, pp 337-347, 1981.

Vijay K. Garg (S'84-M'89) received his Bachelor of Technology degree in computer engineering from the Indian Institute of Technology, Kanpur, in 1984. He continued his education at the University of California, Berkeley where he received his MS in 1985 and Ph.D. in 1988 in Electrical Engineering and Computer Science. He is currently an assistant professor in the Department of Electrical and Computer Engineering at the University of Texas, Austin. His research interests are in the areas of distributed systems and supervisory control of discrete event systems. He has authored or co-authored more than 50 research articles in these areas. He has served as a program committee member of the IEEE International Conference on Distributed Computing Systems and as an organizer of the minisymposium on Discrete Event Systems in the SIAM Conference on Control and Applications.

Joydeep Ghosh (S'83-M'88) received a Ph.D. in Computer Engineering from the University of Southern California in 1988, where he was the first student in the School of Engineering to be awarded an "All-University Predoctoral Merit Fellowship" for four years. He is currently an Associate Professor in the Department of Electrical and Computer Engineering at the University of Texas, Austin. Joydeep's research interests include parallel computer architecture and artificial neural networks, and he has over 50 refereed publications in this areas. He served as the general chairman for the SPIE/SPSE Conference on Image Processing Architectures, Santa Clara, Feb. 1990, and cochair for ANNIE'93. He is a member of the editorial board of IEEE Computer Society Press, and of Pattern Recognition. Dr. Ghosh received the 1992 Darlington Award for best journal paper from IEEE Circuits and Systems Society.