On the Correctness Criteria of Load Balancing Programs

Anish Arora [1]

Computer & Information Science
The Ohio State University
Columbus, OH 43210

Mohamed G. Gouda

Computer Sciences
The University of Texas
Austin, TX 78712

September 28, 1999

Abstract

Load balancing programs are usually verified by analyzing and simulating simple performance models. In this paper, we argue that besides this verification the correctness of load balancing programs needs to be verified formally, in order to gain confidence in the use of these programs in practical situations where the simple models are not always respected. Towards this end, we propose a set of correctness conditions that need to be satisfied by load balancing programs. Moreover, we show that these correctness conditions are not unduly restrictive, by designing a rich family of load balancing programs that satisfy these conditions. The presented programs are distinguished by their properties of full distribution, scalability, adaptivity, fault-tolerance, and guaranteed progress irrespective of the speed at which the environment produces or consumes load.

Keywords: load balancing programs, verification, distribution, constraint, convergence

---

[1] Contact Address: Dreese 395, 2015 Neil Avenue Mall, Columbus, OH 43210, USA
Tel: (614)292-1836 , Fax: (614)292-2911 , Email: anish@cis.ohio-state.edu
Supported in part by NSF Grant CCR-93-08640, OSU Grant 221506 and NSA MDA904-96-1-1011

1 Introduction

Load balancing programs have been the subject of substantial research over the last two decades. The importance of these programs has remained, from the viewpoint of practice, owing to the phenomenal success of low-cost networks of workstations, large-scale information infrastructures containing replicated servers (such as the so called World Wide Web), and multiprocessor architectures that enable fast, concurrent execution of fine-grained threads. From the viewpoint of theory also, these programs remain important, primarily because of the difficulty of verifying a load balancing program, i.e., showing that it improves the performance in a desirable manner.

Given the difficulty of verifying load balancing programs (see Sidebar 1), guarantees about program behavior are typically made under simple, idealized performance models. This is especially true when program behavior is verified only under the conditions of simple probability distributions or trace-driven simulations. The program behavior in actual practice is verified only informally, if it is verified at all. However, informal verification is often misleading for programs that are distributed and time-sensitive. It is possible that the actual program behavior deviates significantly from the predicted behavior if the environment behavior deviates even slightly from the simple assumptions made about it. For instance, if it is optimistically assumed that the network roundtrip communication time would be less than the interarrival time between any two jobs, a simple load balancing program that polls all nodes so as to choose an optimal target node for each job is acceptable, but if in practice the interarrival time becomes slightly less than the roundtrip communication time, the program can be unacceptable.

It follows that it is beneficial to formulate correctness criteria against which load balancing programs can be formally verified. Demonstrating that the correctness criteria are met adds substantially to confidence in the use of a load balancing program. It, moreover, complements the verification provided by analytical or experimental methods.

To better explain the role of correctness criteria, let us consider by way of analogy the problem of routing. In routing, paths between nodes in a network are to be calculated so that messages may be efficiently delivered from source to destination nodes. Verification of routing programs is subject to almost all of the difficulties that verification of load balancing programs is: the environment is not fully predictable and may change over time, faults may occur which change the availability and performance of nodes and channels, the network may be heterogeneous, information at nodes about the network may not be current, etc. Given these difficulties, routing programs do not all guarantee that messages will always be delivered in finite time, just as as load balancing programs do not all guarantee that jobs in input queues will be (executed and) delivered in finite time to output queues. Still, good performance, i.e., efficient delivery, in practical situations is the principal requirement in routing, just as it is in load balancing.

To verify a routing program, the standard approach is not only to guarantee program behavior under strong assumptions, but also to formally verify that the program satisfies some correctness criteria regarding the structure, safety and progress of the program (for instance, that cyclic paths do not persist indefinitely). Thus, program behavior in general is accounted for by formal verification of correctness, while program behavior in ideal situations is predicted by analytical and experimental verification. The same argument can be made for load balancing programs, provided suitable correctness criteria can be identified for them.

Our goal in this paper is therefore to suggest correctness criteria for formally verifying load balancing programs. These correctness criteria guarantee desirable program behavior even if very lax assumptions are made about the behavior of the environment, the topology of the network, and the states of the nodes. In particular, they allow the environment to produce or consume loads at any time in an arbitrary manner. Moreover, these correctness criteria are not unduly restrictive, which we demonstrate by designing various load balancing programs that meet these criteria.

The programs we design are further distinguished by several properties that are desirable in practice. They are fully distributed; thus, they are scalable. Their actions are specified in a parametric form that allows performance to be tuned to suit the current environment behavior, communication pattern, and processor states; thus, they are adaptive. Moreover, they are able to perform in spite of arbitrary changes in the network topology; thus, they are fault-tolerant. Lastly, they avoid divergent or chaotic behavior when executed concurrently with an environment whose speed varies arbitrarily; thus, they guarantee progress and are, to the best of our knowledge, the first such balancing programs proven to do so. The abovementioned properties are especially important for modern day networks and information structures where scale is a pervasive concern, there is wide variation in terms of load, network congestion and failure / repair of processors and channels occur frequently, and transient states are the norm and not the exception.

The rest of the paper is organized as follows. In Section 2, correctness criteria are formalized for verifying load balancing. In Section 3, in accordance with the correctness criteria, a basic program is derived that balances work loads between adjacent processors only. The basic program is augmented, in Section 4, to achieve system-wide load balancing in ring networks; in Section 5, to achieve system-wide load balancing in tree networks; and in Section 6, to achieve system-wide load balancing in directed acyclic networks and, thereby, in any structured or unstructured network. Progress of these programs in the presence of arbitrary interactions with the environment is discussed in Section 7. Finally, concluding remarks are made in Section 8.

2 Correctness Criteria

A load balancing program is formulated in terms of a undirected, connected graph. The nodes represent processors and the edges represent communication channels between processors.

Associated with each node u in the graph is a variable, integer number $x.u$ of load units. The value of each $x.u$ represents the current 'load' of processor u. (See Sidebar 3 for how $x.u$ is chosen.) It is assumed that load units are produced and/or consumed in any number, at any time, and at any node, by the environment of the program.

A computation of a load balancing program is a maximal sequence of program steps: Every step in the sequence executes at some nodes in the graph; it starts execution in the state immediately prior to it and yields a different state upon execution. Maximality of the computation implies that if the sequence of steps is finite then the final step of the computation yields a fixpoint state, i.e., a state that remains unchanged upon execution of any (arbitrarily chosen) program step.

Notice that program computation is allowed to be unfair with respect to the nodes in the graph. It is thus possible to capture programs where the speed of program steps involving some nodes differs arbitrarily from their speed at steps involving other nodes, and changes over time. Notice also that concurrent execution has been captured by interleaving the program steps; this is generally accepted to be a reasonable assumption in the formal verification of concurrent programs.

We are now ready to state the correctness criteria for a load balancing program. A load balancing program is 'correct' iff each computation of the program satisfy the following three conditions.

- *Distribution* : Each step of program computation accesses the load units of a bounded number of adjacent nodes (i.e., nodes in the graph with an edge between them).

- *Constraint* : Each step of program computation can only move load units between nodes, but not produce new units nor consume existing units at any node.
- *Convergence* : Starting from any state (i.e., with arbitrary integer numbers assigned to the x's), program computation is guaranteed to terminate in a finite number of steps in a state where

  $(\forall$ nodes u and v in the graph : $\| x.u - x.v \| \leq 1)$

The distribution criterion specifies the basic limitations of a distributed program: communication is limited to occur between adjacent processors and cannot involve an unbounded numbers of processors. The constraint criterion specifies that the program computation cannot itself produce or consume units, it can only move units between processors; it follows that program steps preserve the sum of all x's in the graph. The convergence criterion specifies that if the environment no longer perturbs (i.e., produces or consumes) the load, the program computation eventually terminates at a state where the load is balanced.

These criteria are minimal in the sense that they omit several implementation-level concerns. For instance, they leave unspecified the "location policy" (that determines which sites initiate the balancing process, send loads, or receive loads), the "transfer policy" (that determines when, which, and how much load to send or receive), and the "information policy" (that determines when and how much of the program state is to be accessed). The omission is intentional: thus, load balancing programs can be verified for various types of implementations and underlying architectures.

3 Designing Local Convergence

We adopt the following two-part approach to designing load balancing programs that meet the correctness criteria formulated in the previous section: first, we design a program computation that balances the work load of adjacent nodes and, subsequently, we augment the design with a means for ensuring that the work load of non-adjacent nodes is also balanced. In this section, we consider the first part.

More specifically, we weaken the correctness criteria stated in Section 2 by replacing the convergence criterion with the following criterion:

> *Local Convergence* : Starting from any state, program computation is guaranteed to terminate in a finite number of steps in a state where
>
> $(\forall$ *adjacent* nodes u and v in the graph : $\| x.u - x.v \| \leq 1)$.

and derive a program that meets the weakened correctness criteria.

From the distribution criterion, a step of program computation has the form

$$\begin{aligned}&\textbf{if } (u \text{ and } v \text{ are adjacent nodes in the graph}) \ \wedge \ B\\ &\textbf{then } x.u\,,\, x.v := F.(x.u, x.v)\,,\, G.(x.u\,, x.v)\end{aligned} \tag{1}$$

It remains to deduce predicate B and functions F and G.

From the local convergence criterion, no step can change the fixpoint state where $|\,x.u{-}x.v\,| \leq 1$. This can be accomplished by choosing B as

$$B \ \equiv \ |\,x.u - x.v\,| > 1$$

Without loss of generality, we assume $x.u \geq x.v$ in which case B can be simplified as

$$B \ \equiv \ x.u - x.v > 1 \tag{2}$$

From the constraint criterion, changing $x.u$ and $x.v$ by F and G should keep their sum fixed. This can be accomplished by choosing F and G as follows.

$$\begin{aligned}&F \text{ decreases } x.u \text{ by some, possibly negative, integer } \triangle \text{ , and}\\ &G \text{ increases } x.v \text{ by the same } \triangle\end{aligned} \tag{3}$$

From (1), (2), and (3), we can rewrite the step of program computation as

$$\begin{aligned}&\textbf{if } (u \text{ and } v \text{ are adjacent nodes in the graph }) \ \wedge \ \ x.u - x.v > 1\\ &\textbf{then } x.u\,,\, x.v := x.u - \triangle\,,\, x.v + \triangle\\ &\textbf{where } \triangle \text{ is any integer satisfying some predicate } C.(\triangle, x.u, x.v)\end{aligned} \tag{4}$$

It remains to deduce the value of $\triangle$ (or, equivalently, to deduce predicate C) so that the computation terminates in a finite number of steps. Note that this requirement (of termination) is the only one that we have not yet used in our design of a step of program computation.

Deducing the value of $\triangle$. To guarantee termination, it is sufficient to find a ranking function r that assigns to each state s of the computation a natural number $r.s$ such that if a step leads the computation from state s_1 to s_2, then

$$r.s_1 > r.s_2 \tag{5}$$

One possibility for r is

$$r = (\text{sum}\ u\ :\ u \text{ is a node in the graph} :\ x.u^2\ ) \tag{6}$$

Now assume that one step is executed changing $x.u$ and $x.v$ into $x.u{-}\triangle$ and $x.v{+}\triangle$ while leaving all other $x.w$'s unchanged. (This implies that $x.u - x.v > 1$ from (4).) To ensure that (5) is satisfied, it is sufficient from (6) to show that

$$x.u^2{+}x.v^2 \ \ > \ \ (x.u{-}\triangle)^2 + \ (x.v{+}\triangle)^2$$

We simplify the last expression.

$$\begin{array}{ll} & x.u^2{+}x.v^2 \ \ > \ \ (x.u{-}\triangle)^2 + \ (x.v{+}\triangle)^2 \\ = & \{\text{arithmetic}\} \\ & 0 \ > \ (-2 \times x.u \times \triangle) \ \ + \ \ (2 \times x.v \times \triangle) \ \ + \ \ (2 \times \triangle^2) \\ = & \{\text{arithmetic}\} \\ & (x.u{-}x.v) \times \triangle \ \ > \ \ \triangle^2 \\ = & \{\text{arithmetic}\} \\ & x.u{-}x.v \ \ > \ \triangle \ > \ 0 \end{array}$$

Thus, (5) is satisfied provided

$$C.(\triangle, x.u, x.v) \ \ \equiv \ \ x.u{-}x.v > \triangle > 0 \tag{7}$$

The step of program computation. From (4) and (7) the computation step can be written as:

if (u and v are adjacent nodes in the graph) $\wedge$ $x.u - x.v > 1$

then $x.u\,,\, x.v := x.u - \triangle\,,\, x.v + \triangle$

where $\triangle$ is any integer satisfying $(x.u - x.v > \triangle > 0)$

Theorem 0 *The load balancing program designed above satisfies the distribution, constraint, and local convergence conditions.*

Note that the program allows $\triangle$ to be chosen differently in different steps. Moreover, each $\triangle$ can be chosen arbitrarily from within a range (cf. (7)) rather than being assigned a pre-determined value (e.g., $(x.u - x.v)/2$). This provides our load balancing program with the flexibility to adapt the computation to suit the environment behavior.

Since the program satisfies the local convergence criterion but not necessarily the convergence criterion, we now proceed to augment the program computation, in each of the next three sections, so as remedy this limitation for special classes of distributed systems, respectively ring, tree, hypercube (and other) networks.

4 Designing Global Convergence in Ring Networks

Observe that the convergence criterion is equivalent to requiring that program computation terminate at a state where $(max\, u\, :\, x.u) - (min\, u\, :\, x.u) \leq 1$ holds. Hence, one strategy to satisfy the convergence criterion, given that the program designed thus far satisfies the local convergence criterion, is to augment that program to guarantee termination at a state where some pair of adjacent nodes are assigned maximum and minimum work loads.

This guarantee is easily implemented in a ring network: Distinguish a node TOP on the ring, and require that the work load of all nodes along one direction of the ring, starting from TOP, is nonincreasing. In other words, require that $x.u \geq x.(N.u)$ for each node u such that $N.u \neq TOP$, where $N.u$ denotes the node adjacent to u in the chosen direction. Recalling the distribution and the constraint criteria and that the program designed thus far satisfies local convergence, this requirement is met by augmenting the step of program computation to swap the values of $x.u$ and $x.(N.u)$ when $x.u{+}1 = x.(N.u) \wedge N.u \neq TOP$ holds.

The net effect of this requirement is that regardless of the starting values of the x variables, if the computation terminates then in its final state TOP has maximum work load, and the node u such that $N.u = TOP$ has minimum work load.

Formally, the augmented step of program computation is:

---

if	(u and v are adjacent nodes in the graph) $\wedge\ \ x.u - x.v > 1$
then	$x.u\ ,\ x.v := x.u{-}\triangle\ ,\ x.v{+}\triangle$ **where** $\triangle$ satisfies $(x.u - x.v > \triangle > 0)$
elseif	$x.u{+}1 = x.(N.u)\ \wedge\ N.u{\neq}TOP$
then	$x.u\ ,\ x.(N.u) := x.(N.u)\ ,\ x.u$

Theorem 1 *The load balancing program for ring networks is correct.*

Proof: We focus on the proof of termination. Note that if executing the step involves executing the first assignment statement, then the value assigned by the function r to the current state decreases, where $r = (\ \mathit{sum}\ u\ :\ u$ is a node in the graph $:\ x.u^2\ )$.

Also, if executing the step involves executing the second assignment statement, then the value assigned by the function r to the current state is unchanged and the value assigned by the function s to the current state decreases, where $s = (\ \mathit{sum}\ u\ : (D.u \times x.u)\ )$ and $D.u$ is the distance between TOP and u in the ring along the chosen direction. To check the decrease, it suffices to note that

$$D.u \times x.u\ +\ (D.u{+}1) \times (x.u{+}1)\quad >\quad (D.u{+}1) \times x.u\ +\ D.u \times (x.u{+}1)\ .$$

Therefore, the lexicographic ranking function t, $t = \langle r\ ,\ s \rangle$, suffices to prove termination. $\Box$

We illustrate this program by an example. Figure 1 shows an execution of the program given a ring network of nodes 0..3, whose TOP is node 0. The initial state of the nodes (Figure 1(a)) results from load productions and consumptions by the environment such that the x values at nodes 0..3 are 1, 18, 8, and 13, respectively. In the first step, nodes 0 and 1 balance using $\triangle = 8$ (Figure 1(b)). In the next step, nodes 2 and 3 balance with $\triangle = 3$, yielding a state that satisfies local —but not global— convergence (Figure 1(c)). Then, nodes 1 and 2 execute a swap to ensure that $x.1 \geq x.2$ (Figure 1(d)). From this last state, nodes 0 and 1 can balance with $\triangle = 1$ to satisfy global convergence (Figure 1(e)).

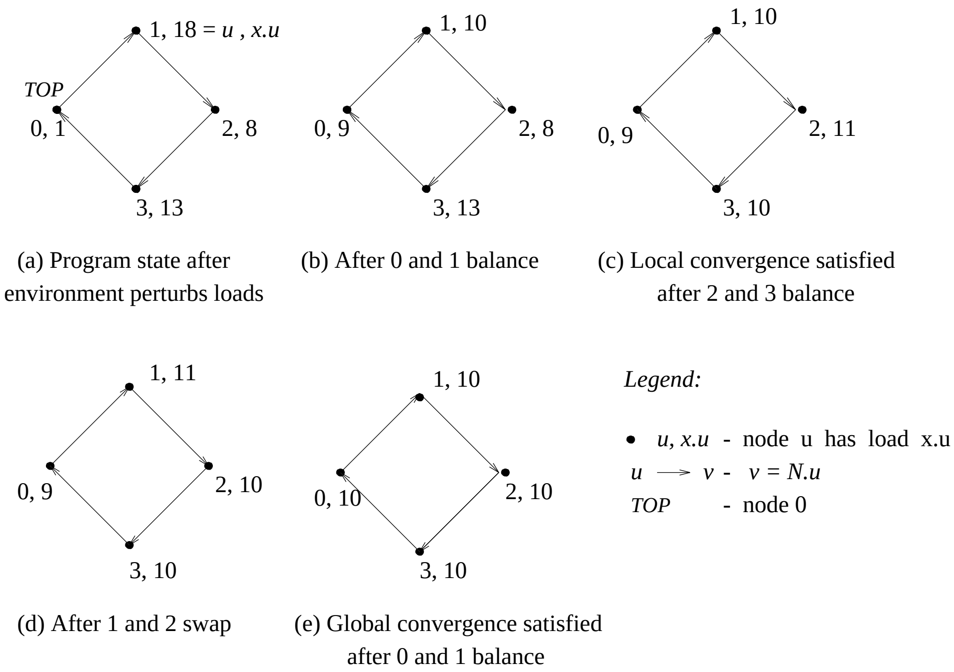

Figure 1: An execution of the load balancing program in ring networks

5 Designing Global Convergence in Tree Networks

An alternative strategy to satisfy the convergence criterion, given that the program designed in Section 3 satisfies the local convergence criterion, is to augment that program to guarantee termination at a state where the node work load along every simple path starting at a distinguished node TOP is (i) nonincreasing and (ii) decreases at most once.

This guarantee is easily implemented in a tree network: Let TOP be the tree root and let $P.u$ be the parent of each node u in the tree. To achieve (i), require that the work load at every node be the maximum of the work load of all nodes in the subtree rooted at that node. Recalling again the distribution and the constraint criteria and that the program designed thus far satisfies local convergence, this requirement is met by swapping the values of $x.u$ and $x.(P.u)$ when $x.u = x.(P.u)+1$ holds.

To achieve (ii), a means of recognizing that node work load along a path starting at TOP decreases more than once is necessary. Therefore, require that each node u maintains a boolean variable $b.u$ so that as long as the work loads of nodes along a path are identical, the b values at those path nodes are maintained to be *true*; if the work load decreases from a path node to its successor, the b value changes from *true* at that node to *false* at its successor; at subsequent nodes in the path, the b value are maintained to be *false*. This requirement is met by ensuring that $b.u$ is always $true$ for $u = TOP$. And, $b.u$ is set to $b.(P.u)$ when $x.u = x.(P.u)$ holds and $false$ when $x.u{+}1 = x.(P.u)$ holds for $u{\neq}TOP$. (The value of $b.u$ is arbitrary when none of these cases apply.) It now follows that if more than one decrease occurs along a path then —assuming the work load along that path is locally balanced and nonincreasing— there exists a node u such $x.u{+}1{=}x.(P.u)$ and $b.(P.u){=}false$ hold. For such u, the values of $x.u$ and $x.(P.u)$ are swapped, so that the smaller work load of u can be subsequently balanced with the larger work load of the parent of $P.u$.

Unfortunately, the program computation designed thus far admits an infinite cycle of swaps, as is illustrated next. Let $x.u = x.(P.u){+}1 = x.(P.P.u){-}1$ and $b.(P.u){=}false$. Now, $x.u$ and $x.(P.u)$ can be first swapped towards achieving (i), and then swapped again towards achieving (ii). This problem is resolved by requiring both swaps to occur *only when* the state of $P.u$ is consistent with its parent, i.e., $x.(P.u){=}x.(P.P.u) \;\wedge\; b.(P.u){=}b.(P.P.u)$ or $x.(P.u){+}1{=}x.(P.P.u) \;\wedge\; \neg b.(P.u) \;\wedge\; b.(P.P.u)$. (We abbreviate this consistency criterion as the predicate $ok.(P.u)$.)

The net effect of these requirements is that regardless of the starting values of the x and b variables, if the program computation terminates then in its final state TOP has the maximum work load, the work load along every path starting at TOP decreases at most once, and every node u satisfies $ok.u$.

The augmented step of program computation is stated below. Statement A propagates a larger work load from u to $P.u$. Statement B propagates a smaller work load from u to $P.u$. Statements C and D maintain the variable $b.u$ of u. (For convenience in stating the computation step, we let $P.TOP$ be TOP itself; hence, $ok.TOP$ is always true. Also, we refer to these statements as the statements of u.)

---

if	(u and v are adjacent nodes in the graph) $\wedge\; x.u{-}x.v > 1$	
then	$x.u\,,\,x.v := x.u{-}\triangle\,,\,x.v{+}\triangle$ **where** $\triangle$ satisfies $(x.u{-}x.v > \triangle > 0)$	
elseif	$x.u{=}x.(P.u){+}1 \;\wedge\; ok.(P.u)$	
then	$x.u\,,\,x.(P.u)\,,\,b.(P.u) := x.(P.u)\,,\,x.u\,,\,true$	(A)
elseif	$x.u{+}1{=}x.(P.u) \;\wedge\; \neg b.(P.u) \;\wedge\; ok.(P.u)$	
then	$x.u\,,\,x.(P.u) := x.(P.u)\,,\,x.u$	(B)

elseif	$x.u = x.(P.u) \;\wedge\; b.u \not\equiv b.(P.u)$	
then	$b.u := b.(P.u)$	(C)
elseif	$x.u + 1 = x.(P.u) \;\wedge\; b.(P.u) \;\wedge\; b.u$	
then	$b.u := false$	(D)

Theorem 2 *The load balancing program for tree networks is correct.*

Proof: To prove termination, we note that if executing the step involves executing the first assignment statement, then the value assigned by the function r to the current state decreases, where $r = (\ sum\ u\ :\ u$ is a node in the graph : $x.u^2\ )$. Also, executing any other statement does not change the value assigned by r to the current program state.

Hence, it remains to show that every sequence of steps executing only the statements A, B, C, or D is finite. We show this by proving a stronger result: Consider an arbitrary subtree of the given tree. Let S be an arbitrary sequence of steps involving execution of statements A, B, C, and D of nodes that are in the subtree. Then, S is finite.

Our proof is by structural induction on the height of the subtree.

Base Case : Height of the subtree is 0.
In this case, the subtree comprises only one node. By itself, a single node is always balanced; hence, S is the empty sequence.

Induction Step : Height of the subtree exceeds 0.
In this case, let w be the root of the subtree. We claim that there exists a finite prefix of S such that the statements of w are not executed in the corresponding suffix; i.e., there exists a suffix of S that consists of executions of statements of only descendents of w.

To prove the claim, we consider two cases:

- $w = TOP$: in this case, $P.w = w$ and, hence, statements A, B, C and D of w do not execute.
- $w \neq TOP$: in this case, we first show that there exists a finite prefix of S such that in the corresponding suffix the statements A and B of w do not execute; we then show that once the statements A and B of w do not execute, the statements C and D of w together execute at most once.

  Once statement A of w executes, statement B of w is not executed since A establishes $b.(P.w)$. In addition, each time statement A of w updates $x.w$, the value assigned by the function r' to the current state decreases, where $r' = (\ sum\ u :\ \ x.u) - x.(P.w)$. Other suffix steps do not change the value assigned by r' to the current state. Alternatively, if statement A of w is not executed, then each time statement B of w updates $x.w$, the value assigned by the function r'' to the current state decreases, where $r'' = x.(P.w) - (\ min\ u : u \text{ is in the subtree} : x.u)$. Other suffix steps do not increase the value assigned by r'' to the current state unchanged. Hence, eventually statements A and B of w do not execute.

Now, consider a suffix of S in which statements A and B of w do not execute. If C or D of w execute in this suffix, the resulting state satisfies the state predicate $ok.w$. As long as this predicate holds, statements C and D are not executed. The only statement executions that can violate $ok.w$ are of statement A of any child of w —in which case $\neg ok.w\ \wedge\ b.w\ \wedge x.w > x.(P.w)$ holds— and statement B of any child of w —in which case $\neg ok.w\ \wedge\ \neg b.w\ \wedge\ x.w < x.(P.w)$ holds. In the former case, no statement of w or any child of w may be executed thereafter, thereby preserving the predicate $\neg ok.w\ \wedge\ b.w\ \wedge x.w > x.(P.w)$. In the latter case, no statement of w may be executed thereafter, but the statement B (and not A) of a child of w may be executed, which however preserves the predicate $\neg ok.w\ \wedge\ \neg b.w\ \wedge\ x.w < x.(P.w)$. Hence, eventually statements C and D of w do not execute either.

Now, once the statements of w do not execute, the execution of statements A, B, C or D in one subtree of w does not affect the execution of these statements in another subtree of w (note that only the first assignment in the computation step affects the variables of nodes in different subtrees). Hence, based on our claim, we can apply the induction hypothesis for each child of w, to conclude that the suffix is also finite. □

6 Designing Global Convergence in Hypercube, Mesh, and Omega Networks

In this section, we first show that the load balancing program designed for tree networks is readily adapted for directed acyclic networks. Then, we apply the adapted program in structured networks, such as hypercubes, meshes and omega networks, using constant-time embeddings of directed acyclic graphs in these networks. Finally, we compare the strategies of load balancing that embed a directed acyclic graph (henceforth, dag) with those that embed a ring or a tree, and discuss the fault-tolerance of our load balancing schemes.

To adapt the program for tree networks to acyclic networks, we lift two assumptions that we made earlier when we considered tree networks: we lift the assumption that there is a unique distinguished node TOP to allow multiple distinguished root nodes TOP; and we lift the assumption that there is a unique parent for each node to allow multiple parents for each node.

These allowances notwithstanding, our strategy for achieving convergence remains the same as that for tree networks: guarantee termination at a state where (i) the node work load on every path starting at any TOP node is nonincreasing, and (ii) the node work load along every path starting at any TOP decreases at most once.

This guarantee is easily implemented in an acyclic network, by modifying the computation step for tree networks as follows: (a) Statement A is allowed to propagate larger loads to *any* parent of u, (b) Statement B is allowed to propagate smaller loads to *any* parent of u, (c) Statement C maintains variable $b.u$ according to the values of the variables of u and *all* parents of u, and (d) Statement D maintains variable $b.u$ according to the values of the variables of u and *any* parent of u.

Let $Par.u$ be the set of parents of u in the acyclic graph. And let $ok.v =$

$$((\forall w : w \in Par.v : x.v = x.w) \ \wedge \ b.v = (\forall w : w \in Par.v : b.w) \qquad \vee$$
$$((\exists w : w \in Par.v : x.v+1 = x.w) \wedge \neg b.v \wedge (\forall w : w \in Par.v : x.v = x.w \vee (x.v+1 = x.w \ \wedge \ b.w)))$$

More specifically, the step of program computation for tree networks is modified as follows:

Statement A is replaced by one choice for each $v \in Par.u$:

elseif $\quad x.u = x.v+1 \ \wedge \ ok.v$
then $\quad x.u\,,\, x.v\,,\, b.v := x.v\,,\, x.u\,,\, true$

Statement B is replaced by one choice for each $v \in Par.u$:

elseif $\quad x.u+1 = x.v \ \wedge \ \neg b.v \ \wedge \ ok.v$
then $\quad x.u\,,\, x.v := x.v\,,\, x.u$

Statement C is replaced by the choice:

elseif $\quad (\forall v : v \in Par.u : x.u = x.v) \ \wedge \ b.u \not\equiv (\forall v : v \in Par.u : b.v)$
then $\quad b.u := (\forall v : v \in Par.u : b.v)$

Statement D is replaced by one choice for each $v \in Par.u$:

elseif $\quad x.u+1 = x.v \ \wedge \ b.v \ \wedge \ b.u$
then $\quad b.u := false$

Theorem 3 *The load balancing program for acyclic networks is correct.*

Proof: To prove termination of the adapted program computation, we note that if executing the computation step involves executing the first assignment statement, then the value assigned by the function r to the current state decreases, where $r = (\ sum\ u\ :\ u$ is a node in the graph $:\ x.u^2)$. Also, executing any other statement does not change the value assigned by r to the current program state.

Hence, it remains to show that every sequence of steps executing only the statements A, B, C, or D is finite. We show this by proving a stronger result: Consider an arbitrary subgraph of the given graph. Let S be an arbitrary sequence of steps involving execution of statements A, B, C, and D of nodes that are in the subgraph. Then, S is finite.

Our proof is by structural induction on the depth of nodes in the subgraph (where depth is defined as the maximum distance of a node from one of its descendents in the acyclic subgraph). The proof is essentially identical to the one for subtrees in Section 5, and is hence omitted here. $\Box$

Application of Load Balancing in Acyclic Networks to Structured Networks. Each edge in a structured network —such as a hypercube, a mesh, or a butterfly network— is readily directed so that the resulting graph becomes acyclic.

Consider, for instance, a hypercube or a mesh network. If each edge in the network is directed from its incident node with smaller coordinate value to its incident node with larger coordinate value, using a lexicographic ordering, the resulting graph is acyclic. Consider, also, an omega or —more generally— a multistage network. If each edge in the network is directed from its incident node with smaller stage number to its incident node with higher stage number, the resulting graph is again acyclic.

Observe that in both instances the direction of each edge depends only on the coordinates of its incidents nodes and, hence, all edges may be directed concurrently. Hence, embedding a dag requires only a constant amount of time, and load balancing is readily achieved using the adapted computation described above.

Of course, an alternative approach would be to embed a ring or a tree and to use the augmented computation step described in Section 4 or 5, respectively, on the virtual graph. Embedding a ring may, however, yield virtual edges that are dilated, i.e., virtual edges that span multiple edges in the underlying network. Consequently, the resulting load balancing computation may not be fully distributed. Moreover, embedding a ring or a tree may reduce the fault-tolerance of load balancing programs. (Recall that fault-tolerance is one of the stated objectives in the design of load balancing programs.) Thus, whereas the failure or repair of any number of nodes or edges preserves the acyclicity of a dag, the failure or repair of even one node or edge may violate a ring or tree. It follows that to achieve fault-tolerance in ring and tree networks, additional programs are needed to reconfigure the virtual graph in the presence of failures and repairs. Several such programs have appeared in the literature; for example, [1, 2] present reconfiguration programs for trees and rings that tolerate any finite number of failures and repairs.

We conclude this section with the remark that similar considerations occur when designing load balancing computations for arbitrary networks. Again, rings, trees, or dags may be embedded, but the specific choice of which embedding is appropriate depends upon the case at hand.

7 Progress Criteria

Verifying that a load balancing program meets the correctness criteria given in this paper guarantees that no matter how the environment perturbs the state of a program, subsequent computation of the program *in isolation* terminates at a state where the loads are balanced system-wide. In this section, we consider an additional correctness criterion, regarding the progress of load balancing programs when executed *concurrently with* its environment.

Recall that the environment of a load balancing program can both produce new units as well as consume existing units of load. It is convenient to think of the environment steps as perturbations on the x values of the nodes. The environment steps, unlike the program steps, are not constrained: they may perturb the x values arbitrarily.

It is straightforward to observe that the correctness criteria of Section 2 guarantee that if the environment perturbs the state at a speed less than that of program convergence, the program is infinitely often at a balanced state. And, if the environment perturbs the state at arbitrarily high speeds but the durations of such perturbations are finite, then the program starts to converge eventually (cf. [4]).

For environments that perturb that state at arbitrary and variable speeds, to show that the program behaves reasonably, the following correctness criterion may be additionally required:

- *Progress* : Starting from any state, in all interleavings of productions and consumptions of units by the environment and steps of program computation, no unit is infinitely often moved between nodes.

In other words, regardless of the speed of environment perturbations, the amount of these perturbations, and the time duration of these perturbations, these programs move each load unit at most a finite number of times. It follows that the program behavior does not diverge nor thrash when it executes concurrently with any environment.

Theorem 4 *The program for local convergence satisfies progress.*

Proof: Let us conceptually organize the load units at each node in a vertical pile. Thus, we can uniquely associate with each load unit its height in its pile. Let us also ensure that when the environment produces load units, they are placed at the tops of these piles; when the environment consumes load units, they are removed anywhere from these piles; and, when local balancing steps move load units, they are moved from the top of the one pile to the top of the other pile.

Using this organization, perturbations by the environment do not increase (and possibly decrease) the height of any existing load unit. Also, local balancing steps decrease the height of the load units that they move and leave the respective heights of the remaining units unchanged. Hence, regardless of how the environment perturbs the state of the local convergence program during program execution, every step of that program leaves the global state "more" balanced. In other words, each load unit moves only finitely often in any interleaving of program steps and environment perturbations. □

Theorem 5 *The program for global convergence in a ring network satisfies progress.*

Proof: Steps in our program for global balancing in ring networks either decrease or leave unchanged the height of the existing load units that they move. Hence, every program step leaves the global state "at least as" balanced. Moreover, in each step where a load unit is moved but its height does not change, its distance from the TOP node (in the chosen direction) decreases and, hence, the unit can be moved at most M times without decreasing its height (where M is the number of nodes). □

As can be expected, not all load balancing programs that satisfy the correctness criteria of Section 2 also satisfy the additional criterion of progress. By way of example, in the case of our global tree and dag global convergence programs, there exist interleavings of program and environment steps where some load units repeatedly do the following: first, these load units move towards TOP nodes as a result of some environment step and, then, they move away from TOP nodes as a result of some other environment step. In all these moves, the height of the load units remains unchanged. Consequently, the constrained convergence of our tree and dag programs is insufficient to satisfy progress for all environments.

Nonetheless, there do exist tree and dag load balancing programs that satisfy progress as well the other three correctness criteria. Towards exhibiting such programs, we exploit the idea that to avoid having load units that move both towards TOP nodes and away from TOP nodes – without a decrease in their height – we may choose one direction, say the direction towards TOP nodes, such that if any load unit moves in that direction without a decrease in its height, then that unit cannot subsequently move in the other direction.

Based on this idea, we propose that the x load units at each node be divided into two vertical piles of height y and z, respectively. For convenience, we say that two piles are adjacent if they are in the same node or in adjacent nodes. As before, we add local balancing steps between adjacent piles that decrease the height of each load unit they move.

Each y pile load unit remains on a y pile in program steps that do not decrease its height, as follows. Whenever the y pile of a node has one more unit than the y pile for (one of its) parents, its highest unit is moved to the parent. The net effect is that when y pile units move either their height decreases, or their height remains the same and their distance from TOP nodes decreases.

In contrast to y pile units, each z pile unit can move to a y pile or z pile in program steps that do not decrease its height, as follows. Whenever the z pile of a parent node has one more unit than the z pile of some child, its highest unit is moved to the child; also, whenever the z pile of a node has one more unit that its y pile, its highest unit is moved to the y pile. The net effect of these steps is that when z pile units move their height decreases, or their height remains the same and their distance from leaf nodes decreases, or they move to a y pile.

Thus, effectively, in the new tree/dag programs, local load balancing is performed between the y pile and the z pile at each node, between adjacent y piles, and between adjacent z piles. Moreover, z piles are maintained to be nondecreasing in the direction from the TOP nodes to the leaf nodes, the z pile of each node is maintained to be at most its y pile, and the y piles are maintained to be nondecreasing from leaf nodes to TOP nodes. (The reader will note the similarity to the load balancing program for ring networks.) The resulting program satisfies global convergence, as well as progress in all environments.

More formally, the step of program computation in tree networks is (where m and n range over piles):

if	(m and n are adjacent piles) $\wedge$ $m - n > 1$
then	$m\,,\, n := n{-}\triangle\,,\, n{+}\triangle$ **where** $\triangle$ satisfies $(m - n > \triangle > 0)$
elseif	$y.(P.u){+}1 = y.u$
then	$y.u\,,\, y.(P.u) := y.(P.u)\,,\, y.u$
elseif	$z.u{+}1 = z.(P.u)$
then	$z.u\,,\, z.(P.u) := z.(P.u)\,,\, z.u$
elseif	$(y.u{+}1 = z.u \;\wedge\; u{\neq}TOP)$
then	$y.u\,,\, z.u := z.u\,,\, y.u$

The step for program computation in dag networks is similar.

Theorem 2 *The program above for global convergence in tree/dag networks satisfies progress.*

It follows from the progress of our programs, that if the environment consumes the load from the bottom of each load pile at a non-zero speed, then every load unit is consumed eventually.

8 Concluding Remarks

We have suggested correctness criteria against which load balancing programs can be formally verified. By using the correctness criteria, confidence is gained in the use of a load balancing program in practical settings. The criteria are complementary to analytical and experimental methods. We have demonstrated the criteria in a wide variety of load balancing programs, which are further distinguished by their properties of full-distribution, stability, adaptivity, fault-tolerance, and scalability.

Our programs make very lax assumptions about the predictable behavior of the environment. Since they satisfy the convergence condition, it follows that even if the environment arbitrarily perturbs the program only a finite number of times, the load balancing program will eventually converge to a balanced state. Since they satisfy the progress condition, it follows that even if the environment arbitrarily perturbs the programs an infinite number of times, but consumes load units at each node at a non-zero (but otherwise arbitrary) speed, each load unit will eventually be consumed. Thus, we can make predictions about program behavior even if the environment behaves quite unpredictably.

We should emphasize that our correctness criteria are not intended to be exhaustive or unique. It is possible that additional criteria (such as progress) be considered depending upon the application in question. Minor variations of the criteria may also be considered as need be. For instance, while we have chosen to specify loads as integers, it might be more appropriate in some applications to choose a different type for the load variables. Likewise, it might sometimes be appropriate to vary the distribution criterion, say to capture alternative architectures, or the convergence criterion, say to capture different degrees of balancing, such as $(\forall\, u, v \;:\; |\, x.u - x.v \,| \;\leq\; \alpha)$ for some integer α. It is nonetheless important that the criteria be formulated explicitly, as opposed to the usual practise of leaving them implicit, so that formal verification becomes possible.

Note that our programs have not explicitly modeled communication delays. Delays are readily accommodated in our designs by refining the atomicity (i.e., the granularity) of program actions. For instance, a refinement could implement high-atomicity actions that allow a node to instantaneously access the state of neighboring nodes by low-atomicity actions that obtain the state of the neighboring nodes in a one-at-a-time manner and then use local information only to take load balancing decisions. An example refinement of a load balancing program has been presented by Gronning et al [3]. In this example, actions with same atomicity as ours are refined into actions that use message passing in hypercube of Transputers, while preserving local convergence.

Our correctness criteria have been influenced by the notion of self-stabilization [5]. This notion enables programs to withstand perturbations, by guaranteeing recovery of the program to desired execution even after the program is placed in an arbitrary state. Generalizations of this notion have been found useful, for instance, in verifying adaptivity [6] and fault-tolerance [7]. In this paper, we have considered a generalization of self-stabilization that we call constrained convergence. We conclude that constrained convergence is useful and merits further consideration.

Convergence Span. Proofs of the convergence criterion that exhibit variant functions have the added advantage that they provide upper bounds on the convergence span. We measure these bounds in terms of rounds [1]. Intuitively, a round consists of a sequence of steps wherein each node makes some minimal progress. More precisely, a round is a minimal, nonempty sequence of steps wherein for each node there exists a step where the node either executes one of its statements or the if-conditions of all its statements are false before or after the step.

Specifically, from the variant function of the local load balancing program, we can deduce that N^2 rounds is an upper bound on the convergence span, where N is the sum of the x values of all nodes. In this particular case, the bound can be tightened further: Ernie Cohen [8] has shown that the convergence span of our local load balancing program is $\Theta(N^{1.5})$ rounds, in the sense that (i) every local load balancing computation converges within $O(N^{1.5})$ rounds and (ii) there exists a program and a state starting from which a local load balancing computation requires $\Omega(N^{1.5})$ rounds to terminate. Further, we claim that the convergence span of the ring-based program is $O(N^{1.5}\ max\ NM)$, where M is the number of nodes, and that of the tree- and dag-based programs is $O(N^{1.5}\ max\ ND)$ rounds, where D is the depth of the tree or dag at hand.

Future work. Topics that we are studying include: design of load balancing programs that are (i) specialized for structured architectures or (ii) that instead of using a fixed underlying virtual structure, create periodic waves to reach and balance all network nodes. Moreover, we are investigating formal characterizations of the speeds of environment perturbations at which load balancing is better than no load balancing.

References

[1] A. Arora, M. G. Gouda: Distributed reset. IEEE Transactions on Computers **43(9)** (1994) 1026-1038

[2] A. Arora, A. Singhai: Fault-tolerant reconfiguration of trees and rings in networks. Journal of High Integrity Design **1(4)** (1995) 375-384

[3] P. Gronning, T.Q. Nielsen, and H.H Lovengreen: Stepwise development of a distributed load balancing algorithm. Proceedings of the Fifth International Workshop on Distributed Algorithms (1991)

[4] J. A. Stankovik: Stability and distributed scheduling algorithms. IEEE Transactions on Software Engineering **SE-11(10)** (1985) 1141-1152

[5] E. W. Dijkstra: Self-stabilizing systems in spite of distributed control. Communications of the ACM **17(11)** (1974) 643-644

[6] M. G. Gouda, T. Herman: Adaptive programming. IEEE Transactions on Software Engineering **17(9)** (1991) 911-921

[7] A. Arora, M. G. Gouda: Closure and convergence: A foundation of fault-tolerant computing. IEEE Transactions on Software Engineering **19(11)** (1993) 1015-1027

[8] E. Cohen: On the convergence span of greedy load balancing. Information Processing Letters (1994)

SIDEBAR 1 : Why Verification of Load Balancing is Difficult

A key complication in verifying a load balancing program is that its environment is not fully predictable: Arrival and departure rates of loads are difficult to characterize in practice; they tend to be irregular, processor-dependent, and time-dependent. Verification has therefore to consider program behavior not only in the steady-state but also in the transient-state and, in some cases, also the adaptivity of behavior as load rates change. For these behaviors, the performance in the presence of load balancing (say, in terms of the mean or variance of) response time, completion time, queue sizes, etc. has to be shown, and compared with the performance in the absence of load balancing.

Another key complication is distribution. Distribution implies that the state of all processors executing the load balancing program is not directly accessible to any processor. Moreover, owing to communication and computation delays resulting from distribution, indirectly accessed state are out-of-date. This raises verification concerns of the sort 'can underloaded processors inadvertently become overloaded' or 'can some load units be indefinitely exchanged between processors'.

Since load balancing is often motivated by the need for dealing with faults or configuration changes in the processor set, yet another complication in its verification is robustness. Program availability has to be shown in the presence of addition or removal of processors and communication channels during the balancing process.

Finally, the program could also have to be shown to deal with various forms of heterogeneity; for instance, some load types may be preempted while others may not, and some load types may only be executed on certain processors.

SIDEBAR 2 : Traditional Techniques for Verification of Load Balancing

The difficulty of verifying load balancing programs has led to research from many different perspectives [9]. Early work on load balancing concentrated on the static form of the problem in which the (evolving) program state is not used in balancing decisions. Verification was based on models such as graph-theoretic flows, queuing networks, and Markov processes.

By way of contrast, models for dynamic load balancing are more complex than static ones. Experimental and simulation based approaches have often been used to verify the conditions under which dynamic load balancing programs improves performance. Heuristic and expert system models have also been used to justify some load balancing programs.

Stability concerns, e.g. 'can some load units be indefinitely exchanged between processors', are notoriously difficult to verify. Stankovic [4] has studied these concerns from the perspective of convergence of stochastic learning automata. And, Burgess and Passino have studied them from the perspective of Lyapunov stability [10], thereby verifying convergence and its rate in programs that balance continuous and discrete loads even in the presence of bounded delays in sensing and passing loads.

A few efforts have been made that considered some of the correctness criteria suggested in this paper. Lin and Keller [11] and Gronning et al [3] have designed self-stabilizing [5] load programs that satisfy "local convergence"; i.e., they balance loads between neighboring processors, but not necessarily between non-neighboring processors. And, Kam and Bastani [12] have shown a self-stabilizing program that maintains a hierarchical ring in a dynamic network partitions a globally known, static set of load units between the processors in the hierarchical ring.

References

[9] B.A. Shirazi, A.R. Hurson and K.M. Kavi: *Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE Computer Society Press (1995) 1-42

[10] K. L. Burgess and K. M. Passino: Stability analysis of load balancing systems. International Journal of Control, **61(2)** (1995)

[11] F. C. H. Lin, R. Keller: The gradient model load balancing method. IEEE Transactions on Software Engineering **13(1)** (1987) 32-38

[12] M. K. Kam, F. B. Bastani: A self-stabilizing ring protocol for load balancing in distributed real-time process control systems. Technical Report #UH-CS-87-9, Department of Computer Science, University of Houston (1987)

SIDEBAR 3 : Defining the Load

A variety of indices can be used to define the load value x for each node. Typically, the x values are chosen to be a function of one or more of the following indices: node queue length, processor utilization, resource utilization, processor idle time, processing time, response time, completion deadlines, arrival rate of tasks, and service rate of tasks.

Index measurements can be instantaneous or averaged over some interval. It is heuristically argued that the measurements should favor the recent past over the distant past, so as to better predict the near future. Also, to accommodate heterogeneity, the measurements should be scaled to a virtual value to favor the execution of certain loads on certain nodes.

The choice of the index depends upon the performance objective of the load balancing. For example, Lin and Raghavendra [16] have observed that the load sharing objective of minimizing average task response time is achieved by balancing x values that denote the server idle times in a weighted least-square sense.

In several cases, it is found that a simple choice of index is sufficient. Ferrari and Zhou [14] and Kunz [15] have reported experiments where choosing only node queue length performs better than other choices of single-dimensional and multidimensional indices.

While it is generally prefered that load indices be nonnegative, it is worthwhile to note that our programs satisfy the correctness criteria even if the x values are negative. Note also that the x values do not include the overhead of executing the load balancing program as part of the x loads. It follows that the cost-benefit analysis of load balancing is favorable only if the benefit achieved by the load balancing is measurably more than the cost of executing of the load balancing program itself. Formal verification is not intended to supplant this analysis; traditional techniques will suffice for this purpose.

References

[14] D. Ferrari and S. Zhou: A dynamic load balancing policy with a central job dispatcher (LBC). Proceedings of Performance'87, Twelveth International Symposium on Computer Performance Modeling, Measurement, and Evaluation (1987) 515-528

[15] T. Kunz: The influence of different workload descriptions on a heuristic load balancing scheme. IEEE Transactions on Software Engineering **17(7)** (1991) 725-730

[16] H.-C. Lin and C.S. Raghavendra: A dynamic load balancing policy with a central job dispatcher (LBC). IEEE Transactions on Software Engineering **18(2)** (1992) 148–158