Brief Announcement: Sorting on Skip Chains*

Ajoy K. Datta[1], Stéphane Devismes[2], and Lawrence L. Larmore[1]

[1] School of Computer Science, University of Nevada Las Vegas, USA
[2] VERIMAG UMR 5104, Université Joseph Fourier

1 Introduction

Sorting values on a chain of processes is a well-known problem, and a number of algorithms has been published [1,2]. We consider here a generalization of this problem, where the processes that have values, called *major processes*, are separated from each other by any number of intermediate processes, called *relay processes*, which do not have their own values, although they can read and write the major values while doing their job of relaying those values.

More precisely, we consider a chain network of n processes. Some of those processes, including the two end processes, are *major* processes, and the rest are *relay* processes. We call this structure a *skip chain*. The problem is then to sort the values held by the major processes. We call this problem the *skip chain sorting problem*.

We propose a silent self-stabilizing distributed algorithm for the skip chain sorting problem. Our algorithm is written in the locally shared memory model and works under an unfair daemon. Its stabilization time is $O(md)$ rounds, where m is the number of major processes and d is the maximum number of processes in the chain from one major process to the next. Note that $md = O(n)$ if the spacing between major processes is roughly equal.

2 Formal Statement of the Problem

We are given a chain of processes. Some of those processes, including the two end processes (which we call *L* and *R*) are *major* processes, and the rest are *relay* processes. We call this structure a *skip chain*. We assume that only major processes have values, and the problem is to sort those values. The specification of the skip chain sorting problem is given below.

1. In an arbitrary configuration of a skip chain, there is a *canonical value* $V(x)$ associated with each major process x. This value may or may not be stored at x.
2. At each step, the multiset of canonical values does not change, although the canonical values of two different major processes can be exchanged.
3. Every computation eventually results in a *legitimate configuration*, where the following conditions hold:
   (a) The canonical values of the major processes are in increasing order from left to right.
   (b) The canonical value of each major process x is stored at x.
   (c) No action is enabled.

* The full version of this paper is available at `tinyurl.com/3dydywq`. This work has been partially supported by the ANR project *ARESA*2.

X. Défago, F. Petit, and V. Villain (Eds.): SSS 2011, LNCS 6976, pp. 443–444, 2011.
© Springer-Verlag Berlin Heidelberg 2011

We assume that n is the number of processes in the chain, m is the number of major processes, and d is the *relay chain length*, which is the maximum number of processes in the chain from one major process to the next. For example, if all processes are major processes, then $d = 2$, and $d = n$ if only the two end processes are major.

3 Overview of the Solution

We give an algorithm, *skip chain sort* (SCS) essentially a distributed version of the well-known algorithm *bubblesort*, which satisfies the requirements listed above.

SCS is self-stabilizing, which implies that it converges to a legitimate configuration regardless of the initial configuration. Given any skip chain $\mathcal{S}$, let $\mathcal{C}$ be the set of all configurations of SCS on $\mathcal{S}$. A certain subset $\mathcal{N} \subseteq \mathcal{C}$ consists of what we call *normal* configurations. These configurations are those where the states of all processes are correct, except that the canonical values may not be sorted. $\mathcal{N}$ is closed under the actions of SCS and is an attractor of $\mathcal{C}$.

The first phase of SCS, which we call *error correction*, results in a normal configuration. The second phase of SCS sorts the canonical values of the major processes, and eventually halts in a *legitimate* configuration, where each major process stores its own canonical value, and no process is enabled to execute.

Every major process x, except L, contains two embedded relay processes, which we call x.*l_relay* and x.*r_relay* (at the end, each major node stores its canonical value in its right relay); L contains only one embedded relay process, L.*r_relay*. We call the other relay processes *free relay processes*. If x is any process, then we define *Right_Major*(x) and *Left_Major*(x) to be the nearest major processes to the right and left of x (if any) respectively.

If x is a major process, we define the *right relay chain* of x to be the chain of relay processes starting with x.*r_relay* and ending with *Right_Major*(x).*l_relay*; the *left relay chain* of x is simply defined to be the right relay chain of *Left_Major*(x).

Two values can only be swapped by a major process if it holds both. If x is a major process and $y =$ *Right_Major*(x), then $V(x)$ and $V(y)$ can be compared, and possibly swapped, by y. The mechanism is to move $V(x)$ along the right relay chain of x to y.*l_relay*, while $V(y)$ is at y.*r_relay*. The values are then compared and possibly swapped. Afterward, the new value of $V(x)$ can move back to x, while the new value of $V(y)$ can move to *Right_Major*(y). After at most $\binom{n}{2}$ such comparisons, the canonical values will be sorted.

SCS uses *color waves* to control the movement of the values along the relay chains. A value moves to the left at the crest of a wave of color 0, and to the right at the crest of a wave of color 1. Two additional colors, 2 and 3, complete the color wave cycle to avoid ambiguity between waves. Additionally, there is an "error color," **E**.

When the canonical values are sorted, a *silence wave*, generated by the rightmost process, moves to the left, eventually causing all execution to cease.

References

1. Flocchini, P., Kranakis, E., Krizanc, D., Luccio, F., Santoro, N.: Sorting and election in anonymous asynchronous rings. Journal of Parallel and Distributed Computing 64, 254–265 (2004)
2. Sasaki, A.: A time-optimal distributed sorting algorithm on a line network. Information Processing Letters 83, 21–26 (2002)