Convergence of Causal Message Ordering within the Schiper-Eggli-Sandoz Algorithm

Christian D. Newman
Department of Computer Science
Kent State University
Kent, Ohio, USA
cnewman@kent.edu

Abstract**—** *The convergence of the Schiper-Eggli-Sandoz Causal ordering of messages is measured in order to verify the scalability of the algorithm under increased number of processes and unfavorable message load.*

Keywords-Causal Message Ordering; Distributed Algorithms.

I. Introduction

The Schiper-Eggli-Sandoz algorithm deals with causal message ordering in a system that has multiple, asynchronous processes. In order to guarantee causal message receipt, it takes advantage of piggy-backing information on messages and keeping information at each process about the other processes. If a message cannot be delivered due to being causally preceded, it's buffered; messages that are buffered remain buffered until the messages that causally precede it are received. The amount of time these messages remain buffered for increased number of processors will give us an idea of how well the algorithm scales.

II. Base work

The base algorithm used in this research was created by Schiper, Eggli, and Sandoz[1]. Their algorithm is explained, in detail, within their paper but I will summarize its use here. The algorithm is used for causal ordering of message receipt at each process within a system. In order to guarantee causal receipt, it uses a vector at each process to store the ID of each previously sent message to any other process within the system. This vector of previously sent messages is appended to each message that is sent from the current process and, when this message arrives at its destination process, the process checks this vector to make certain that there are no messages propagating that causally precede the current message. If there are messages propagating that causally precede the current message, then the current message is buffered.

III. Convergence Of Causal Message Ordering

The Schiper-Eggli-Sandoz algorithm buffers messages that are causally preceded by other messages that have not been received. The research in this paper seeks to measure the behavior of the unbuffering of these messages in both average-case and worst-case computations of this algorithm. In understanding this behavior, we may be able to find ways to further optimize the way the algorithm runs, the way messages are sent by processes or, at the very least, conclude that the Schiper-Eggli-Sandoz algorithm is or is not scalable.

IV. Experiment Setup

For the purpose of this experiment, an implementation of Schiper-Eggli-Sandoz's algorithm was written in c++. There is a single channel that the processes all share and the engine is responsible for picking random messages from the channel to send to each process, causing that process to use its receive guarded command. Two message sending schemes were created: The first causes each process to send its messages in the opposite order in which they should be received. This represents the worst-case computation where each process must buffer n-1 of the messages it receives (the last message is the one it should have received first in a list of n messages). The second sending scheme is a random scheme where the processes send randomly by inserting messages into the channel destined for randomly selected receivers. This represents the average-case computation, where message receipt order is non-deterministic in nature. To verify the algorithm, console output recorded the data within the previously sent vector of the message and the current time at the receiving process. Manual comparisons were made for multiple runs of the algorithm to insure that it works just as in Schiper-Eggli-Sandoz's paper. After algorithm verification, the algorithm was modified to measure the average number of messages that were buffered before a message ended up being unbuffered. It did this continuously until the computation reached a fix point. The point of convergence is when the average number of messages buffered before a message is unbuffered, is one. This is because, at this point, every message that was buffered has been unbuffered (hence why the average comes out to 1). Tests were run on increments of 5 processes each sending 20 messages, all of the way up to 100 processes.

V. Results

Predictably, the average case performed better than the worst-case. There was, however, no other conclusive data that could be obtained from the graph of the average case (see figure 1). The performance of the algorithm neither deteriorated or became better as more processes were added. It only varied within the range of values between 2.9 and 3.5.

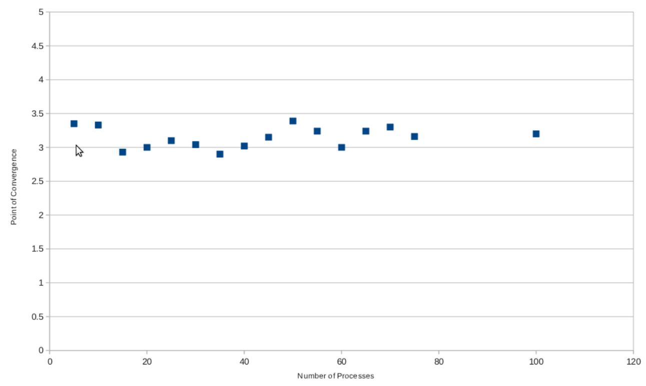

Figure 1. Convergence Results with Random Receive. In figure 1, the results of the tests don't correlate with deterioration or improvement on the part of the algorithm for increased or decreased numbers of processes.

The interesting results came when tests were ran with the reverse-order send scheme: It seems to tend towards the average-case's complexity as the number of processes increase (Figure 2).

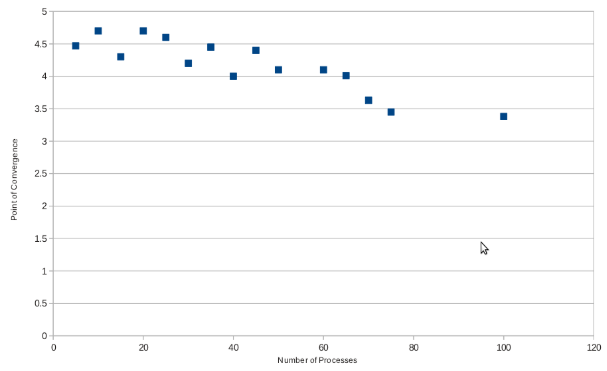

Figure 2. Convergence results with reverse-order receive. The results of the tests correlate with faster convergence for increased numbers of processes.

The results in figure 2 are interesting because it shows that the algorithm actually scales rather well with increased number of processes in the worst-case scenario. The reason we see this type of scalability is actually directly related to how messages are unbuffered. When a message is unbuffered, the buffer is checked again to make certain that, by unbuffering the previous message, no other message can be unbuffered. Since messages are given to processes at random, the increased number of processes makes it more likely that you find a message that causally precedes other messages in the buffer (or the message that was just received). Furthermore, when a message that causally precedes other messages is delivered, large numbers of messages are likely to be delivered all at once. Consider N messages in the channel in reverse-order:

N, N-1, N-2, N-3, …, 3, 2, 1

Every message from N down to message 2 will have to be buffered. However, as soon as message 1 is received, every message from 2 to N will be unbuffered in sequence before any other messages are sent or received. This, as well as the fact that a higher number of processes increases the chance that you receive a message that causally precedes others, is why large numbers of messages all end up unbuffered simultaneously under the worst-case scheme and, also, part of the reason why the worst-case scheme tends towards the average-case's complexity.

VI. Future Work

A larger number of tests with a larger number of processes would give even more accurate answers, especially when it comes to the reverse-order message scheme since, as noted, it seemed to tend towards the average-case with higher numbers of processes. This research leveraged the average in order to measure the behavior of algorithm's unbuffer feature. The average is not a robust measure and, so, outlying datapoints may heavily effect the outcome of the measurements made. A more robust measure might be required to be results that are less inclined to be corrupted by bad datapoints. This algorithm, when run on a sequential machine as opposed to some sort of cluster or distributed environment, is rather memory-heavy. It would be easier to run larger tests on actual distributed systems.

References

[1] André Schiper, Jorge Eggli, Alain Sandoz: A New Algorithm to Implement Causal Ordering. WDAG 1989: 219-232