Evaluation Criteria for Process Synchronization

R. J. Lipton,† L. Snyder++
and Y. Zalcstein+++ 

Research Report #90

† Department of Computer Science, Yale University, New Haven, Connecticut 06520. Part of this work was done while at IBM Research Center at Yorktown Heights, and part was supported by ONR under grant N00014-75-C-0752.

++ Department of Computer Science, Yale University, New Haven, Connecticut 06520. Supported, in part, by ONR under grant N00014-75-C-0752.

+++ Supported, in part, by NSF grant DCR75-01998.
EVALUATION CRITERIA FOR PROCESS SYNCHRONIZATION

R. J. Lipton
L. Snyder
Computer Science Department
Yale University
New Haven, Connecticut 06520

Y. Zalcstein
Computer Science Department
State University of New York
Stony Brook, New York 11794

Abstract — While there are by now well—established criteria for evaluating serial algorithms, such as space and time measures, these criteria cannot be readily applied to asynchronous algorithms. We propose a method for the evaluation of the performance of an asynchronous algorithm. This method is based on the study of delays that are often introduced when one solves a synchronization problem. We then illustrate this method by proving results about the efficiency of various solutions to synchronization problems.

1. Introduction

A central problem in computer science is that of evaluating competing systems for the same task. In the case that the algorithms are to be executed sequentially, several evaluation criteria are commonly used. First, it is easy to express the idea that two algorithms "do the same thing" by the requirement that they have the same input—output behavior. Secondly, given that two algorithms have the same input—output behavior, they may be compared by considering the execution time required, memory space required, numerical (or other type of) stability and so on. By contrast asynchronous algorithms cannot be evaluated so easily, due to several important reasons.

First, asynchronous algorithms — especially those used in operating systems — are not necessarily supposed to halt. Indeed, considerable effort is often required to guarantee that they do not halt, i.e. do not deadlock or crash. Therefore, it often makes no sense to discuss the input—output behavior of these asynchronous algorithms. Thus it is not at all clear when two such algorithms "do the same thing".

Another difficulty is that of measuring efficiency. Simply counting the number of steps required to accomplish a task does not reflect the utilization of multiple processors. Are algorithms requiring more steps -- which can be done in parallel -- to be preferred over those requiring fewer steps -- which cannot be done in parallel? Similarly, algorithms requiring less memory are not clearly superior if referencing this memory causes processor interference.

Since we are mainly concerned with synchronization, the questions of efficiency can be stated as: How much overhead is required (and how much is acceptable) to accomplish process synchronization? Will the method we chose to solve our synchronization problem cause delays or interference which are unacceptable?

In this paper we present a criterion for evaluating asynchronous algorithms. Rather than attempt to assign absolute measures of resource utilization — a task that may well be impossible to do in a useful way — we define, relative to a suitable measure of time, for each non-negative integer k, a relation, simulate, between asynchronous algorithms. For asynchronous algorithms Q and P

Q simulate P

will mean that there is a mapping from computations (state changes) in Q to computations in P. This consideration of state changes avoids the difficulty of non—halting algorithms not being input—output comparable. The efficiency of this correspondence (i.e. the amount of overhead Q requires to accomplish the same effect as P) is measured by the integer k. k measures how closely the "parallelism" of Q and P are related. When k = 0, Q uses multiprocessors as efficiently as P, but as k = +∞ Q uses multiprocessors less and less efficiently. Thus there will be a sequence of relations simulate₀, simulate₁, ... which allow increasing freedom with a corresponding decrease in efficiency.

2. The Model

We have used a more "program oriented" model to study related problems ([5] — [8]). However, experience has shown that, as far as the analysis of synchronization is concerned, it is possible to abstract the model further to the language theoretic one which we present below.
The model will ignore such issues as what kind of language the algorithm is specified in, how the actual scheduling is determined, and, most importantly, how the algorithms are actually implemented. These are, of course, important considerations, but it is our contention that a study of the logical implementation of asynchronous programs is of prime importance.

Let \( I \) be a finite set. Elements of \( I \) will be thought of as actions (instructions or statements). Informally, a computation is any sequence of actions that respects the control flow of an asynchronous program \( P \) (we assume fixed initial values of all variables so that different sequences represent true asynchronous behavior and are not merely a reflection of different inputs to \( P \)).

Clearly, if \( x \) is a computation, then so is any prefix (initial subsequence) of \( x \). We formalize this notion as follows.

**Definition:** Let \( I \) be a finite non-empty set. An asynchronous program is a subset \( \sigma \subseteq I^\omega \), the set of all sequences of elements of \( I \), which is closed under the operation of taking prefixes. Elements of \( I \) are called actions and elements of \( \sigma \) are called computations.

**Definition:** Let \( \sigma \subseteq I^\omega \) be an asynchronous program. A cost function is a function \( c: I^\omega \rightarrow \mathbb{N} \), where \( \mathbb{N} \) is the set of non-negative integers which is additive with respect to concatenation, i.e., \( c(xy) = c(x) + c(y) \). Intuitively, \( c \) measures "time".

Let \( \sigma \subseteq I^\omega \) be an asynchronous program and \( c \) be a cost function. Define a delay functionistor\( d_c: I^\omega \times I^\omega \rightarrow \mathbb{N} \) by

\[
d_c(x,f) = \min \{ c(y): yc \in \sigma \text{ and } xyf \in \sigma \}, \text{ where } d_c(x,x) = 0 \text{ if there is no such } y. \text{ If } c(x) = \text{length}(x) \text{ we will denote } d_c \text{ by } d.
\]

\(d_c(x,f)\) measures the minimal amount of "time" as measured by \( c \) that must elapse before \( f \) can execute following \( x \). This quantity is important for several reasons. In a real time system, the value of \( d_c(x,f) \) may be critical to the correctness of the system. Also, given additional structure in the model, the delay function acts as a quantitative measure of how well multiprocessors can be utilized.

When comparing two asynchronous programs \( \sigma \subseteq I^\omega \) and \( \tau \subseteq I^\omega \), it is convenient to think of one of them, say \( \tau \), as implementing the effect of \( \sigma \) by using more primitive operations. According to this view, \( \tau \) is the "compiled" or "macro expanded" version of \( \sigma \). One can then consider a mapping \( \Phi \) from \( \sigma \) to \( \tau \) representing this compilation process. In the model presented here, it will be more convenient to consider the "inverse", say \( \Psi \), of \( \Phi \) from \( \tau \) to \( \sigma \).

Thus a sequence of actions of \( \Psi \) will be the "expansion" of a single action \( g \) in \( \sigma \). The action \( f \) will be considered to implement the action \( g \) while \( a \) and \( b \) will be considered as bookkeeping operations or overhead.

Our model also requires that \( h \) be a homomorphism which simply means that flow of control in \( \tau \) is a copy of the flow of control of \( \sigma \).

Formalizing the discussion above, we obtain the following.

**Definition:** Let \( \tau \) and \( \sigma \) be asynchronous programs, i.e., \( \tau \subseteq \mathbb{E}^\omega \) and \( \sigma \subseteq \mathbb{E}^\omega \). Then \( h \) is a decoder from \( \tau \) to \( \sigma \) provided \( h \) is a string morphism from \( \mathbb{E}^\omega \) into \( \mathbb{E}^\omega \), i.e., \( h(xy) = h(x)h(y) \) for all \( x,y \in \mathbb{E}^\omega \). Clearly, if \( h(x) \) is an empty string and \( h(Q) = \mathbb{E}^\omega \), it is called observable if \( h(f) \neq h(g) \) whenever \( g \neq f \), otherwise it is called a bookkeeping action.

We can now define \( \text{simulate} \).

**Definition:** Let \( Q \) and \( P \) be asynchronous programs over alphabets \( \mathbb{E}^q \) and \( \mathbb{E}^p \) respectively, and let \( c \) be a cost function on \( \mathbb{E}^q \). Then \( Q \) simulates \( P \) provided there is a decoder \( h \) from \( Q \) to \( P \) such that for all \( xeQ \) and \( xeP \), \( f \) observable,

\[
h(x)h(f)eP \text{ implies } d_c(x,f) \leq k.
\]

Intuitively, if \( g \) is simulated by \( h(f) \), then \( g \) is not "stopped" i.e., some action \( g \) may proceed, then \( h(f) \) may proceed at a faster rate, in the sense that there is a bound \( k \) on the amount of time, as measured by \( c \), that must elapse before the action \( f \) corresponding to \( g \) can be "released". This is our measure of efficiency.

The smallest \( k \) such that \( Q \) simulates \( P \) will be denoted by \( \text{delay}(Q,P) \).

3. Examples and discussion

In this section we illustrate the preceding definitions by examples from the literature. To simplify the discussion, we will use a suggestive informal notation, as is commonly employed in the synchronization literature. It should be pointed out that the advantage of the abstract definition of an asynchronous program is its conceptual economy and aid in simplifying proofs. For describing particular examples, a "program oriented" notation is clearly preferable. This is
Consider the following asynchronous programs which we take as defining the semantics of the "first reader–writer problem" of [1] (for a discussion of the semantics of synchronization problems see [5], [6]).

We will now study the relationship between $Q_1$ and $P$. First let $h$ be the mapping defined by:

$$h(c_i) = c_i$$

$$h(e_i) = e_i$$

$$h(h_i) = h_i$$

$$h(j) = j$$

$$h(k) = k$$

$$h(l) = l$$

$$h(x) = \Lambda$$ for all other actions $x$.

It is not difficult to verify that $h(q) = P$. For instance the computation

$$A, B, C, D, A, B, C, D$$

maps under $h$ to $c_1 c_2$.

We wish to measure the efficiency of this solution. First, we claim that for the given decoder $h$, $Q_1$ simulates $P$ implies $k \geq 3$. To see this, take $x = A_i B_j C_k D_l$ and $f = C_2$, then the shortest $y$ such that $xyfQ_1$ is $D_1 A_2 B_2$, which exits from the critical section of reader–1 restores the semaphore $M$ to 1 and then enters the critical section $A_2 D_2$ of reader–2. Thus $d(x, f) = 3$, while $h(x)h(f) = c_1 c_2 e_2$. By a straightforward analysis of cases, based on the observation that one need execute at most three actions between two "successive" observables, it follows that, for this decoder, $k \geq 3$. Next, we claim that under no decoder $h_1$, can either $A_1$ or $B_1$ be observable actions. Assume the contrary and let $h_1(A_1) = c_1$. Since $A_1 B_q Q_1$ and since clearly $h_1(j) = j$, $h(A_1 J) = h(A_1)h(d) = c_1 c_1 \notin P$, which is a contradiction since $h$ maps computations into computations. Similarly, $h_1(B_1) \neq c_1$. Thus each $h_1(C_1) = c_1$ for all $i$ and we can argue as before that $k \geq 3$, or, for some $i$, $h_1(D_i) = c_1$, but, in the latter case $d(A_i, D_i) = |A_i B_i C_i| = 3$ so $k \geq 3$ in all cases. Hence delay $(Q_1, P) = 3$.

Thus $Q_1$ introduces new delays, but delay $(Q_1, P)$ is fixed, independently of the number $n$ of readers.

Let us now compare $Q_1$ with an alternative solution, $Q_2$, represented in figure 2.
Initially "simulate" as being too strong, based on the intuitive feeling that $P$ is "universal". Using the "simulate" relation, we now show that $P$ is "universal" in the sense that for any asynchronous program $F$, there is a $P$ program $Q$ such that $Q$ simulate $F$, for some $k$. However, $k$ grows unboundedly as a function of the size of $P$.

In the following, we will use the $\text{when...do}$ notation introduced in [5]:

$$\text{when } B \text{ do } \theta$$

where $\theta$ is a predicate and $\theta$ is a statement means that $\theta$ is executed only if $B$ is true. Otherwise, control is interrupted until such time as $B$ is true.

Definition: A $P$ asynchronous program is an asynchronous program $P$ such that there is a distinguished subset $A$ of the program variables (the elements of $A$ are called semaphores) which can only be used by actions of the form $P(S)$ or $V(S)$, $S \in A \cup \{0\}$ where

$P(S)$ is \text{when } S > 0 \text{ do } S = S - 1$ and

$V(S)$ is \text{when true do } S + 1$

Theorem 1. For every asynchronous program $P$, there exists a non-negative integer $k$ and a $P$ asynchronous program $Q$ such that $Q$ simulate $F$, with length as cost function.

Proof. Let $P$ be an asynchronous program and suppose $P$ consists of $n$ actions of the form

$$\text{when } B_i \text{ do } \theta_i$$

with respect to this cost function, delay $(Q, P)$ is still $3$. However, delay $(Q, P) \geq n + t$ since for the above values of $x$ and $f$, the program has to go through the "write" section in order to release the reader.

1. Existence Theorems

In this section we give proofs of various simulation results concerning Dijkstra's $P$ and $V$ primitives.

In our previous work [5], [6], [8], we have shown that with respect to a suitable notion of "simulate", $P$ systems are too weak and cannot simulate even rather simple synchronization problems. Many readers of our work objected to "simulate" as being too strong, based on the intuitive feeling that $P$ is "universal". Using the "simulate," relation, we now show that $P$ is "universal" in the sense that for any asynchronous program $P$, there is a $P$ program $Q$ such that $Q$ simulate $P$, for some $k$. However, $k$ grows unboundedly as a function of the size of $P$. 

In the following, we will use the $\text{when...do}$ notation introduced in [5]:

$$\text{when } B \text{ do } \theta$$

where $\theta$ is a predicate and $\theta$ is a statement means that $\theta$ is executed only if $B$ is true. Otherwise, control is interrupted until such time as $B$ is true.

Definition: A $P$ asynchronous program is an asynchronous program $P$ such that there is a distinguished subset $A$ of the program variables (the elements of $A$ are called semaphores) which can only be used by actions of the form $P(S)$ or $V(S)$, $S \in A \cup \{0\}$ where

$P(S)$ is \text{when } S > 0 \text{ do } S = S - 1$ and

$V(S)$ is \text{when true do } S + 1$

Theorem 1. For every asynchronous program $P$, there exists a non-negative integer $k$ and a $P$ asynchronous program $Q$ such that $Q$ simulate $P$, with length as cost function.

Proof. Let $P$ be an asynchronous program and suppose $P$ consists of $n$ actions of the form

$$\text{when } B_i \text{ do } \theta_i$$

with respect to this cost function, delay $(Q, P)$ is still $3$. However, delay $(Q, P) \geq n + t$ since for the above values of $x$ and $f$, the program has to go through the "write" section in order to release the reader.

1. Existence Theorems

In this section we give proofs of various simulation results concerning Dijkstra's $P$ and $V$ primitives.

In our previous work [5], [6], [8], we have shown that with respect to a suitable notion of "simulate", $P$ systems are too weak and cannot simulate even rather simple synchronization problems. Many readers of our work objected to "simulate" as being too strong, based on the intuitive feeling that $P$ is "universal". Using the "simulate," relation, we now show that $P$ is "universal" in the sense that for any asynchronous program $P$, there is a $P$ program $Q$ such that $Q$ simulate $P$, for some $k$. However, $k$ grows unboundedly as a function of the size of $P$. 

In the following, we will use the $\text{when...do}$ notation introduced in [5]:

$$\text{when } B \text{ do } \theta$$

where $\theta$ is a predicate and $\theta$ is a statement means that $\theta$ is executed only if $B$ is true. Otherwise, control is interrupted until such time as $B$ is true.

Definition: A $P$ asynchronous program is an asynchronous program $P$ such that there is a distinguished subset $A$ of the program variables (the elements of $A$ are called semaphores) which can only be used by actions of the form $P(S)$ or $V(S)$, $S \in A \cup \{0\}$ where

$P(S)$ is \text{when } S > 0 \text{ do } S = S - 1$ and

$V(S)$ is \text{when true do } S + 1$
process-i

(1) $L_1:$

\[ P(S_1) \]

(2) $P(S)$

(3) $v + w + 1$

(4) if $v = np$ then $V(E)$

(5) $V(S)$

(6) $P(E)$

(7) if ok = 1 then

(8) $\theta_1$

(9) $ok = 0$

(10) $w + w - 1$

(11) if $v > 0$ then $V(E)$

(12) else $V(M)$

(13) goto $L_1$;

Where $S$ is a mutual exclusion semaphore (initial value 1) used to protect the critical section (2)-(5), $E$ is a semaphore (initial value 0) used to release all actions that may execute at a given step (the "ready-set" in the terminology of [5]). $M$ is a semaphore (initial value 1) used for communication with the monitor. $S_i$ is a local semaphore (initial value 0) which is enabled at a given step if the i-th action may execute at that step. The variable $w$ is a counter, $np$ is a variable giving the number of actions that may execute at any step and $ok$ is a flag.

The monitor process is given by

\[ \text{IM: } P(M) \]

\[ ok = 1 \]

\[ np = \phi(s_1, \ldots, s_n) \]

\[ t = \phi(s_1, \ldots, s_n) \]

\[ \text{if } q(t, 1) \text{ then } V(S_1) \]

\[ \vdots \]

\[ \text{if } q(t, n) \text{ then } V(S_n) \]

\[ \text{goto IM;} \]

$q$ is a function that computes the number of processes that may execute given the values of the $S_i$'s and $\phi$ is a function that figures out which processes may execute and encodes this information into $t$. $q(t, i)$ will decode $t$ and enable $S_i$ accordingly.

The monitor starts and enables some process-i then enters its critical section (2)-(5). Inside the critical section it acknowledges that it is ready to execute and waits on (6) for release. If all pending processes have so acknowledged, one of them is enabled in line (4). Note that the scheduling responsibility has not been usurped by this simulation in the sense of deciding which process will execute next.

Assuming process $j$ is the first to execute beyond line (6), and since ok will be 1, it will execute $\theta_j$, disable all others from executing (9) (since executing $\theta_j$ could have changed which processes may now proceed), acknowledges that it has passed (10) and then releases another process if not all have been released (11), otherwise it releases the monitor (12).

Let $h((8)_1) = 1$ and $h(f) = \lambda$ for all other actions in $Q$. Evidently $h(q) = P$. To bound the efficiency of the simulation, observe that between any two consecutive observable actions $(8)_i$ and $(8)_j$, $r$ bookkeeping actions are executed, where

\[ r < 5u + 2n + 8 + 5v \]

where $u$ is the number of processes that may execute after $(8)_i$ and $v$ the number that may execute after $(8)_j$. Since $u, v \leq n$, $k$ is bounded by $12n + 8$.

The proof of Theorem 1 suggests another cost function -- the number of observable actions in a word. Let us denote this cost function by $c_1$. Then we have the following:

**Corollary.** For every asynchronous program $F$, there exists an PV asynchronous program $Q$ such that $Q$ simulates $F$ with cost function $c_1$.

**Proof.** Immediate from the proof of Theorem 1, since there are only bookkeeping actions between $(8)_i$ and $(8)_j$.

In [8], we have shown that PV systems with only binary ((0, 1)-valued) semaphores are strictly weaker than PV systems in the sense that there are PV systems that cannot be simulated by any PV system with only binary semaphores. However, we have the following:

**Theorem 2.** For every PV asynchronous program $F$, there is a PV asynchronous program $Q$ with only binary semaphores such that $Q$ simulates $F$, with length as cost function.
Proof. The construction is essentially sketched in [2]. For each semaphore $S$ in $P$, add a new mutual exclusion semaphore $E$ (initial value 1) and a new integer variable $x$ (initial value 0). $Q$ is obtained from $P$ by replacing each $P(S)$ by

$$P(E)
\begin{align*}
x &+ x - 1 \\
&\text{if } x < 0 \text{ then begin } V(E); P(S) \text{ end} \\
&\text{else } V(E); \\
\end{align*}
$V(S)$ by

$$P(E)
\begin{align*}
x &+ x + 1 \\
&\text{if } x \leq 0 \text{ then } V(S) \\
&V(E) \\
\end{align*}

Let the third line in each expansion be the observable action. Then clearly $h(x)h(f) \in P$ implies $d(x,f) \leq 4$.

Acknowledgements

We would like to thank M. Condry, E. Ekanadham, P. Henderson, N. Pippenger, and A. Silberschatz for several helpful discussions.

References


Richard J. Lipton  
Lawrence Snyder  
Y. Zalcstein

Yale University  
Department of Computer Science  
10 Hillhouse Ave., New Haven, CT 06520

Office of Naval Research  
Information Systems Program  
Arlington, Virginia 22217

While there are by now well-established criteria for evaluating serial algorithms, such as space and time measures, these criteria cannot be readily applied to asynchronous algorithms. We propose a method for the evaluation of the performance of an asynchronous algorithm. This method is based on the study of delays that are often introduced when one solves a synchronization problem. We then illustrate this method by proving results about the efficiency of various solutions to synchronization problems.