A Simple Proof of an Algorithm for Synchronous Ordering of Messages

Vijay K. Garg

TR-PDS-1998-013 **November 1998**

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

A Simple Proof of an Algorithm for Synchronous Ordering of Messages

Vijay K. Garg*
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712.

November 19, 1998

Abstract

I present a simple proof for an algorithm by Murty and Garg for Synchronous ordering of messages.

1 Introduction

Synchronous ordering is a stronger requirement than causal ordering. A computation satisfies synchronous ordering of messages if it is equivalent to a computation in which all messages are instantaneous. Fig. 1 given an example of a computation which does not satisfy synchronous ordering. For an example of a synchronously ordered computation see Fig. 2.

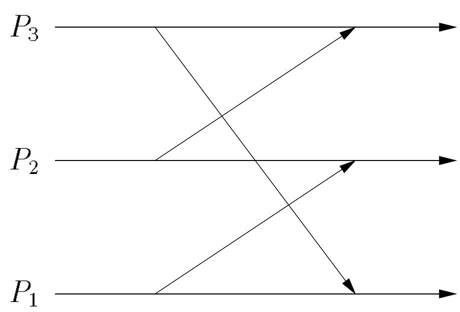

Figure 1: A computation that is not synchronously ordered

Algorithms for synchronous systems are easier to design than those for causally ordered systems. We will see some examples later. It should be observed that as we move up the hierarchy we lose concurrency but make it easier to solve problems.

*supported in part by the NSF Grants ECS-9414780, CCR-9520540, TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant. http://maple.ece.utexas.edu

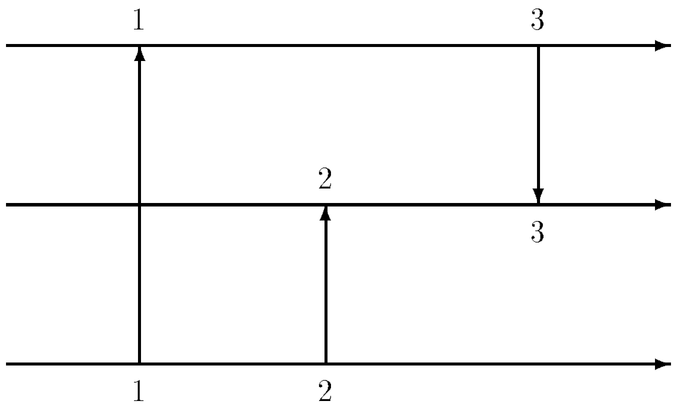

Figure 2: A computation that is synchronously ordered

2 Relationship among Message Orderings

A computation is synchronous if its time diagram can be drawn such that all message arrows are vertical. That is, all external events can be assigned a timestamp such that time increases within a single process and for any message its send and receive are assigned the same timestamp. Formally,

$$\begin{array}{rcl} \exists \mathrm{T} : \mathcal{E} \longrightarrow \mathbb{N} & : & \forall s, r, e, f \in \mathcal{E} \setminus \mathcal{I} \\ s \rightsquigarrow r & \Rightarrow & \mathrm{T}(s) = \mathrm{T}(r) \wedge \\ e \prec f & \Rightarrow & \mathrm{T}(e) < \mathrm{T}(f). \end{array} \qquad \text{(SYNC)}$$

From now on, we assume that there are no internal events, since they do not affect the message ordering; therefore, $\mathcal{E} = \mathcal{R} \cup \mathcal{S}$. It is easy to see that, for any two events e and f

$$(e \rightarrow f) \wedge \neg(e \rightsquigarrow f) \quad \Rightarrow \quad T(e) < \mathrm{T}(f). \tag{1}$$

The following theorem describes the hierarchy of various message orderings.

Theorem 1 *The hierarchy associated with the various modes of communications is*

Synchronous $\subseteq$ *Causally Ordered* $\subseteq$ *FIFO* $\subseteq$ *Asynchronous.*

Proof:

FIFO $\subseteq$ Asynchronous : obvious.

Causally Ordered $\subseteq$ FIFO : This is true because,

$$s_1 \prec s_2 \quad \Rightarrow \quad s_1 \rightarrow s_2.$$

Synchronous $\subseteq$ Causally Ordered : We show that if a computation is synchronous then it is also causally ordered. Since the communication is synchronous there exists a function T satisfying SYNC.

For any set of events $s_1, s_2 \in \mathcal{S}$ and $r_1, r_2 \in \mathcal{R}$ such that $s_1 \rightsquigarrow r_1$, $s_2 \rightsquigarrow r_2$ and $s_1 \rightarrow s_2$:

$$\mathrm{T}(s_1) = \mathrm{T}(r_1) \quad \mathrm{T}(s_2) = \mathrm{T}(r_2), \quad \text{and} \quad \mathrm{T}(s_1) < \mathrm{T}(s_2).$$

It follows that $T(r_1) < T(r_2)$. Therefore, (1) implies

$$\neg(r_2 \rightarrow r_1).$$

$\blacksquare$

The model of synchronous message passing lets us reason about a distributed program under the assumption that messages are instantaneous or "points" rather then "intervals" (i.e., we can always draw the time diagrams for the distributed programs with the message arrows being vertical. If we assume messages as points instead of intervals we can order the messages as a partial order (therefore, we can have vector clocks with respect to messages) satisfying the partial order of the associated events under causality relation. One of the applications for synchronous ordering of messages is that it enables us to reason about distributed objects as if there were centralized. Consider a system with three distributed sites $\mathcal{A}, \mathcal{B}, \mathcal{C}$, each with a set of processes and a set of objects. In specific, let the sites A and B have objects queue 1, and queue 2 respectively. Let there be two processes p_1 at site B and p_2 at site C, such that:

	p_1		p_2
1:	insert (queue 1, c)	1:	insert (queue 2, a)
2:	insert (queue 2, d)	2:	insert (queue 1, b).

If we examine the state of the system of two queues, in a shared-memory model under all possible interleaving, only the following three states are possible:

queue 1 queue 2	c b d a	c b a d	b c a d
Interleaving	$p_1:1$ $p_1:2$ $p_2:1$ $p_2:2$	all other executions	$p_2:1$ $p_2:2$ $p_1:1$ $p_1:2$

In an asynchronous model of message passing the state of the system can be inconsistent, as shown in figure 3. In this case at site A: c is inserted after b in queue 1, and at site B: a is inserted after d, resulting in the following inconsistent state:

queue 1	b	c
queue 2	d	a

If we consider each message as a point, and draw all possible time diagrams with vertical messages, it will result only in one of the consistent states (see figure 4). If we observe the time diagram as shown in figure 3, we notice that, it cannot be redrawn with all messages vertical.

2.1 Crowns in a Distributed Computation

A computation can also be characterized as synchronous based on absence of a structure called "crown". We now present the definition of a crown and prove that a computation is synchronous if and only if it does not contain any crown.

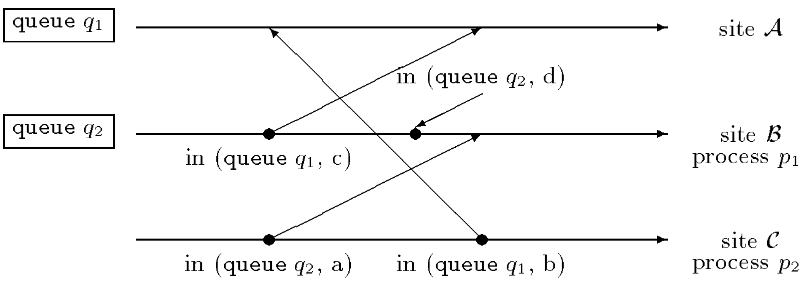

Figure 3: Inconsistent Memory State

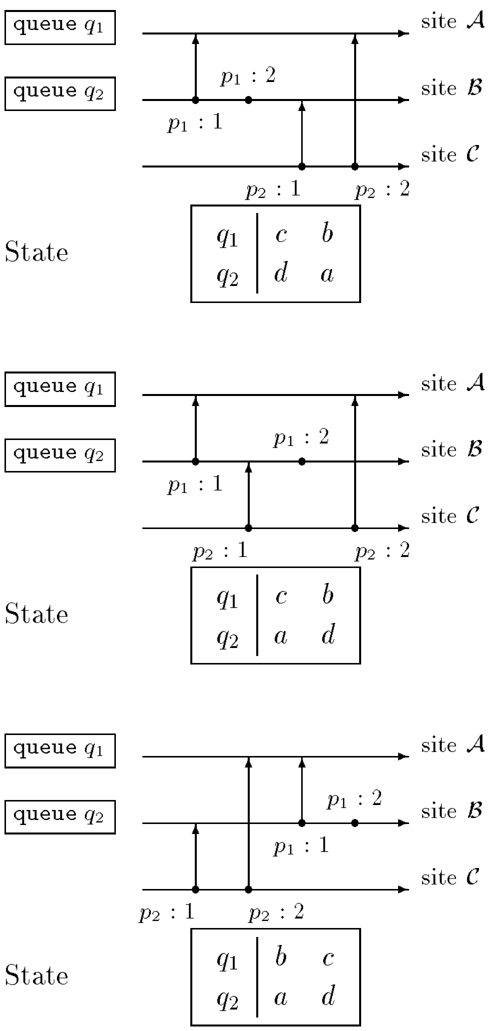

Figure 4: Time Diagrams under Synchronous Message Passing

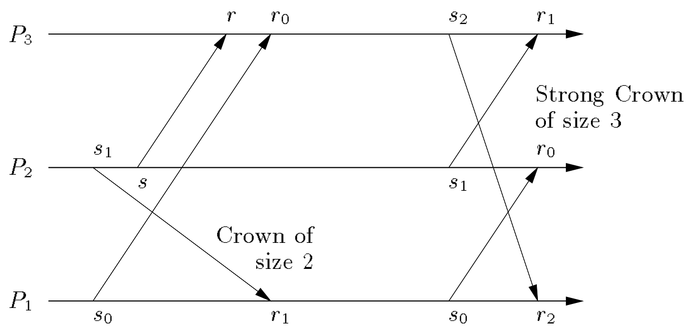

Figure 5: Crowns of size 2 and 3

Definition (Crown): *Let C be a computation. A* crown *(of size k) is a sequence $\langle (s_i, r_i), i \in \{0, 1 \ldots, k-1\} : s_i \rightsquigarrow r_i \rangle$ of pairs of corresponding send and receive events such that (see Figure 5)*

$$s_0 \to r_1, s_1 \to r_2, \cdots, s_{k-2} \to r_{k-1}, s_{k-1} \to r_0.$$

Theorem 2 *A computation is synchronous iff there is no crown in it.*

Proof:

Synchronous $\Rightarrow$ $\neg$Crown :

Since the computation is synchronous there exists a function T satisfying SYNC, and for any two events e and f such that, $(e \to f) \wedge \neg(e \rightsquigarrow f)$ then $\mathrm{T}(e) < \mathrm{T}(f)$.

Suppose, if possible, the computation has a crown of size k, $s_0 \to r_1, \cdots, s_{k-1} \to r_0 \Rightarrow$

$$\begin{array}{lll} \forall i \in \{0, 1, \ldots, k-1\} & \mathrm{T}(s_i) < \mathrm{T}(r_{(i+1) \bmod k}) & (*) \\ \forall i \in \{1, 2, \ldots, k-1\} & \mathrm{T}(s_i) = \mathrm{T}(r_i). & (**) \end{array}$$

Therefore, from equations (*) and (**), $T(s_0) < T(r_0)$. which is a contradiction because SYNC implies that $T(s_0) = T(r_0)$.

$\neg$Crown $\Rightarrow$ Synchronous :

Given a computation, we form a directed graph $G(V, E)$, as follows. The vertex set V consists of all messages in the computation. That is, $v_i = \{s_i, r_i\}$. There is an edge from v_i to v_j if there is an event $e \in v_i$ and an event $f \in v_j$ such that $e \to f$. Thus, $(v_i, v_j) \in E$ iff $(s_i \to s_j) \vee (s_i \to r_j) \vee (r_i \to s_j) \vee (r_i \to r_j)$. It is easy to see that each of the four disjuncts implies $s_i \to r_j$. Hence, $(v_i, v_j) \in E$ iff $s_i \to r_j$.

Since the computation does not have any crown, it follows that the graph G is acyclic and can be topologically sorted. Therefore, there exists a function T: $\mathcal{E} \to \mathbf{N}$ satisfying SYNC.

■

2.2 Algorithm

The algorithm implements the desired ordering using control messages. Note that control messages are not required to satisfy synchronous ordering. In order to distinguish control messages, we use $\mathcal{E}$ to denote events that satisfy synchronous ordering and $\mathcal{E}^c$ to denote send and receive of control messages.

The algorithm shown in Fig. 6 is for the process P_i. All processes have the same algorithm. Observe that the protocol to implement synchronous message ordering cannot be completely symmetric. If two processes desire to send messages to each other then there is no symmetric synchronous computation which allows this - one of them must succeed before the other. Thus to introduce asymmetry we use process numbers to totally order all processes. Each send event s is either from a bigger process (denoted by $big(s)$) or from a smaller process (denoted by $small(s)$). We assume that processes do not send messages to themselves.

In our algorithm, a process can be in two states - active or passive. Every process is initially active. We first consider the algorithm for a send event corresponding to $big(s)$. A process is allowed to send a message to a smaller process only when it is active. After sending the message it turns passive until it receives an *ack* message from the receiver of the message. While passive it cannot send any other message nor can it accept any other message. Thus, the following condition is satisfied.

$$\forall e \in \mathcal{E} : (s \prec e) \wedge big(s) \Rightarrow (r \rightarrow e)$$

It follows that

$$\forall e \in \mathcal{E} : (s \rightarrow e) \wedge big(s) \Rightarrow (r \rightarrow e) \tag{2}$$

Note that the protocol for a message from a bigger process requires only one control message (*ack*).

To send to a bigger process, say P_j, P_i first needs permission from P_j. It can request the permission at any time. P_j can grant permission only when it is active. Further, after granting the permission, P_j turns passive till it receives the message for which it has granted the permission. Thus, the protocol for a message from a smaller process requires two control messages (*request* and *permission*). Since the process that grants the permission turns passive till it receives the message, we get that

$$\forall e \in \mathcal{E} : (e \prec r) \wedge small(s) \Rightarrow (e \rightarrow s)$$

It follows that

$$\forall e \in \mathcal{E} : (e \rightarrow r) \wedge small(s) \Rightarrow (e \rightarrow s) \tag{3}$$

By taking contrapositive and using asymmetry of $\rightarrow$, we also get

$$\forall e \in \mathcal{E} : (s \rightarrow e) \wedge small(s) \Rightarrow \neg(e \rightarrow r) \tag{4}$$

2.3 Proof of correctness

The correctness of the above algorithm is proved in the usual two steps: safety and liveness.

Theorem 3 (Safety) *Any distributed computation that satisfies equations 2 and 3 is synchronous.*

P_i ::

var

state : {active, passive} initially active

To Send a message m to P_j, $(j < i)$

enabled if (state = active)

send m to P_j

state = passive;

Receive m from P_j, $(j > i)$

enabled if (state = active)

send *ack* to P_j

Receive *ack*

state = active

To Send a message (mid, m) to P_j, $(j > i)$

send *request(mid)* to P_j

Receive *request(mid)* from P_j, $(j < i)$

enabled if (state = active)

send *permission(mid)* to P_j

state = passive;

Receive *permission(mid)* from P_j, $(j > i)$

enabled if (state = active)

send m to P_j

Receive m from P_j, $(j < i)$

state = active

Figure 6: Algorithm to Implement Synchronous Ordering of Messages

Proof: To prove that the computation is synchronous we will show that it does not have any crown. We first show that if the computation has any crown of size greater than two, then it also has one of size two.

Pick any message s_i in the crown. If s_i is big, then from $(s_i \to r_{i+1})$ and $big(s_i)$, and equation 2, we get that $r_i \to r_{i+1}$. Combining this with $(s_{i-1} \to r_i)$, we get $(s_{i-1} \to r_{i+1})$. Thus, a smaller crown can be obtained by dropping the message i.

Now consider the case when s_i is small. Then, from $(s_{i-1} \to r_i)$ and $small(s_i)$, and equation 3, we get that $s_{i-1} \to s_i$. Combining this with $(s_i \to r_{i+1})$, we get $(s_{i-1} \to r_{i+1})$. Thus, a smaller crown can be obtained by dropping the message i.

We now show that there cannot exist any crown of size 2. Assume if possible there exists two messages such that $(s_1 \to r_2) \wedge (s_2 \to r_1)$. First consider the case when one of them is big. Without loss of generality let $big(s_1)$ be true. This implies $(r_1 \to r_2)$ from equation 2 by using r_2 for e. Now if $big(s_2)$ holds we get that $(r_2 \to r_1)$ and if $small(s_2)$ holds then we get $\neg(r_1 \to r_2)$ (by applying equation 4). Both cases contradict $(r_1 \to r_2)$.

Now consider the case when $small(s_1)$ and $small(s_2)$ are true. From equation 3, we get $s_2 \to s_1$ as well as $s_1 \to s_2$. Thus, this case is also not possible. Thus, we conclude that no crown is possible.

■

Theorem 4 (Liveness) *In a distributed computation (that implements the algorithm), every process P_k that wants to send a message will eventually be able to send it and every message will eventually be delivered.*

Proof: The smallest process P_1 is always *active*. It sends a reply as soon as it gets a message. Now to apply induction, given that k smallest processes will eventually be in the *active* state, we show that the $(k+1)$th process if *passive* will eventually turn *active*. P_{k+1} can be passive only if it has sent a message or a permission to a smaller process. In either case, since all smaller processes are active it will get a reply and thus become active.

■