Phase Synchronization

Jayadev Misra*
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
(512) 471-9547
misra@cs.utexas.edu

April 27, 1999

1 Problem Statement

Consider a set of asynchronous processes where each process executes a sequence of *phases*; a process begins its next phase only upon completion of its previous phase. What constitutes a phase is irrelevant for our purposes. It is required to design a synchronization mechanism that guarantees that:

- No process begins phase $(k+1)$ until all processes have completed phase k, $k \geq 0$. (Initially, all processes have completed their phase 0.)
- No process will be permanently blocked from executing its phase $(k+1)$ if all processes have completed their phase k, $k \geq 0$.

Assume that the processes communicate through shared variables; contentions for access (read or write) to a shared variable by different processes are resolved arbitrarily but fairly (i.e., any process attempting to read/write a shared variable will do so eventually). *Nothing may be assumed about the initial values of the shared variables.* In the absence of this requirement, the following simple algorithm suffices: A counter variable c is initially 0; c is incremented by 1 whenever a process completes a phase; a process begins its phase $(k+1)$ only if $c \geq k \times N$, where N is the number of processes. One of the applications of phase synchronization is to initialize the variables of a multiprocess system before *any* variable is read, where different processes initialize different portions of the shared store. Here, initialization may be thought of as the first phase and regular computation as the second phase. In order to solve such problems, we assume nothing about the initial values of shared variables.

Phase synchronization arises in a variety of problems (in addition to the shared store initialization problem described above). It is a basic paradigm for constructing synchronous systems out of asynchronous components: A PRAM, for instance, consists of processes that read a common store, compute, and write the common store in one step; steps are synchronized in the sense that no process begins its next step until all processes have completed their current step. Another application of phase synchronization is to abort a computation if a process detects a condition under which the computation should be aborted; it simply does not complete its current phase, thus preventing the remaining processes from starting their next phase. It is easy to take global snapshots [1] or system

*This work was partially supported by ONR Contract N00014-90-J-1640 and by Texas Advanced Research Program grant 003658–065.

checkpoints for a synchronized computation: Upon detecting a global "checkpoint" condition every process executes a checkpoint phase, and upon completion of this phase, it resumes its regular computation; from the synchronization conditions it follows that no process starts its checkpoint phase until all processes have detected the checkpoint condition—and thus frozen their regular computations—and no process resumes its regular computation while some process is still taking a checkpoint.

Phase synchronization can be solved using a general synchronization algorithm; see Chapter 14 of [2]. In this paper we show a very simple scheme for solving this problem.

2 An Informal Description of the Solution

We propose a solution and prove its correctness in informal terms. The solution is developed through a series of refinements.

2.1 A Simple Solution

We introduce an array n of integers that is shared among the processes; element $n.i$ corresponds to process i. Roughly speaking, process i will set $n.i$ to k upon completing phase k, and a process enters its phase $(k+1)$ only after detecting that every $n.i$ is at least k. This basic protocol has to be modified because a process that is attempting to enter its second phase may detect $n.i \geq 1$, though process i has not yet begun its first phase—in this case, $n.i$ happens to be positive initially.

The synchronization protocol for process i is as follows. Initially it sets all $n.j$ to 0 in arbitrary order. After completion of its phase k, $k \geq 1$, process i checks if all $n.j$ are at least k; if such is the case it begins its phase $(k+1)$; otherwise, it sets $n.i$ to k. Note that the checking of $n.j$'s and the setting of $n.i$ may have to be repeated several times.

Protocol for process i

- Initially, set each $n.j$ to 0 in arbitrary order.
- Upon completion of the phase k, $k \geq 1$, execute
  do $\langle \exists\ j\ ::\ n.j < k \rangle\ \rightarrow\ n.i\ :=\ k$ **od**

The test $\langle \exists\ j\ ::\ n.j < k \rangle$ requires simultaneous access to all $n.j$. The next refinement of the solution (in Section 2.3) removes this requirement.

2.2 Correctness of the Simple Solution

We show that

- (Synchronization) No process begins its phase $(k+1)$, $k \geq 0$, until all processes complete their phase k, and
- (Progress) If all processes have completed their phase k, $k \geq 0$, each process will enter its phase $(k+1)$.

Since all processes have completed their phase 0 initially, these two facts hold for $k = 0$. In the following proofs, $k > 0$.

2.2.1 Synchronization

For processes i, j, we show that process i does not enter its phase $(k+1)$ until process j completes its phase k . Process i sets $n.j$ to 0 initially and detects $n.j \geq k$ before entering its phase $(k+1)$. The only process that sets $n.j$ to a positive value—note $k > 0$—is process j, and process j sets $n.j$ to values lower than k before completing its phase k . Therefore, process j must have completed phase k for $n.j \geq k$ to hold.

2.2.2 Progress

Suppose that all processes have completed their phase k, $k > 0$. We show that every process will begin its phase $(k+1)$ eventually. When all processes have completed their phase k either (1) $\langle \forall\ j\ ::\ n.j \geq k \rangle$ holds or (2) $\langle \exists\ j\ ::\ n.j < k \rangle$ holds, in which case every process i will keep setting $n.i$ to k. Once $n.i \geq k$ holds it holds forever because (1) no process other than i assigns to $n.i$ after completing its phase k, $k > 0$, and (2) process i writes k or a higher value into $n.i$ after completing its phase k . Therefore, eventually $\langle \forall\ j\ ::\ n.j \geq k \rangle$ holds and continues to hold. Hence, every process will terminate its loop (after the phase k) and begin the phase $(k+1)$.

Note: It is not sufficient for process i to set $n.i$ to k only once. In particular, after completion of its first phase, process i may set $n.i$ to 1, but later $n.i$ may be set to 0 during initialization by another process. We show later that it is sufficient for process i to set $n.i$ to k once provided k exceeds 1.

2.3 Checking the Shared Variables Asynchronously

2.3.1 A Refined Solution

In the solution of Section 2.1, a process tests all $n.j$ simultaneously. We show that these variables may be tested one by one; once a process detects that $n.j \geq k$, for some j, the process need not test $n.j$ again before entering its phase $(k+1)$. We introduce a local variable $s.i$ of process i that contains a set of process indices; it remains for process i to test $n.j$ for all j in $s.i$. The protocol for process i upon completing its phase k, $k > 0$, is as follows.

$s.i$:= set of all process indices;
do $\langle [\!]\ j\ ::$
$\quad j \in s.i\ \wedge\ n.j < k\ \rightarrow\ n.i\ :=\ k$
$\quad [\!]\ j \in s.i\ \wedge\ n.j \geq k\ \rightarrow\ s.i, n.i\ :=\ s.i - \{j\}, k$
$\rangle$
od

Note on notation: We use "$[\!]\ j\ ::$" to denote that the body of the loop—until "$\rangle$"—should be instantiated over all process indices. Thus, the loop contains two guarded commands for each process. The loop terminates when $s.i$ is empty, since all the guards are then *false*.

2.3.2 Proof of Synchronization

We show that, at any point after process i assigns to $s.i$ upon completing its phase k and before it enters its phase $(k+1)$, all processes outside $s.i$ have completed their phase k . The assignment to $s.i$ (of all process indices), after process i completes its phase k, establishes this assertion. Now, process i sets $n.j$ to 0 initially; no process other than j sets $n.j$ to a positive value—note, $k > 0$—and process j sets $n.j$ to k, or a higher value, only after completing its phase k . Thus, if process i detects $n.j \geq k$ for some j in $s.i$, then process j has completed its phase k . Process i enters its phase $(k+1)$ only when $s.i$ is empty (to see this, take the conjunction of the negation of all the guards in the loop). Hence, all processes have completed phase k by then.

2.3.3 Proof of Progress

After all processes complete their phase k, a process i that is executing its loop will set $n.i$ to k and $n.i$ will never be reset to a lower value. Therefore, $\langle \forall\ j\ ::\ n.j \geq k \rangle$ will hold eventually and continue to hold thereafter. Therefore every process will eventually complete its loop.

2.4 Setting $n.i$ only once, for $k > 1$

Variable $n.i$ need be set only once following completion of phase k by process i, for $k > 1$ (but not $k = 1$). To see this, note that when process i completes its phase k all processes have completed their phase $(k + 1)$; if $(k - 1) \geq 1$—i.e., $k > 1$—no process other than process i will write into $n.i$. Hence process i need write into $n.i$ only once because this value will not be overwritten by another process. Thus, the protocol for process i upon completion of its phase k, $k > 1$, is

$s.i$:= set of all process indices; $n.i := k$;
do $\langle [\!] \; j \;\; ::$
$\quad j \in s.i \;\wedge\; n.j < k \;\rightarrow$ skip
$\quad [\!] \; j \in s.i \;\wedge\; n.j \geq k \;\rightarrow\; s.i := s.i - \{j\}$
$\rangle$
od

Note that this optimization does not apply for $k = 1$ because processes other than process i write into $n.i$ upon completion of their phase 0 (i.e., during their initializations).

2.5 A Shortcut in Testing

Suppose process i determines following its phase k, $k > 0$, that $n.j > k$, for some process j. Then clearly process j has completed its phase $(k + 1)$. Process j could not have begun its phase $(k + 1)$ until all processes completed their phase k . Therefore, process i can then assert that all processes have completed their phase k . This observation leads to the following protocol for process i upon completing its phase k, $k \geq 1$.

- upon completion of phase 1 execute
  $s.i$:= set of all process indices;
  do $\langle [\!] \; j \;\; ::$
  $\quad j \in s.i \;\wedge\; n.j < 1 \rightarrow n.i := 1$
  $\quad [\!] \; j \in s.i \;\wedge\; n.j = 1 \rightarrow s.i, n.i := s.i - \{j\}, 1$
  $\quad [\!] \; j \in s.i \;\wedge\; n.j > 1 \rightarrow s.i := \phi$
  $\rangle$
  od

- upon completion of the phase k, $k > 1$, execute
  $s.i$:= set of all process indices; $n.i := k$;
  do $\langle [\!] \; j \;\; ::$
  $\quad j \in s.i \;\wedge\; n.j < k \rightarrow$ skip
  $\quad [\!] \; j \in s.i \;\wedge\; n.j = k \rightarrow s.i := s.i - \{j\}$
  $\quad [\!] \; j \in s.i \;\wedge\; n.j > k \rightarrow s.i := \phi$
  $\rangle$
  od

2.6 Computing Modulo 3

The given solution requires the elements of n to assume successively larger values as the phase numbers increase with the progress of computation. We show that all $n.i$ can be computed in modulo 3 arithmetic.

Note that whenever process i tests $n.j$ after completing its phase k, $k > 0$,

$$k - 1 \leq n.j \leq k + 1.$$

To see the upper bound, since process i has not begun its phase $(k + 1)$—and hence, it has not completed its phase $(k + 1)$—process j has not begun (nor completed) its phase $(k + 2)$; therefore,

$n.j \leq k+1$. Also, when process k started its phase k, $k > 1$, it detected $n.j \geq (k-1)$ prior to it, and since all processes have completed their first phase, $n.j$ does not decrease thereafter. For $k = 1$, $n.j \geq (k-1) = 0$, because process i must have initialized $n.j$ to 0 and no process sets $n.j$ to a negative value.

Since $n.j$ can assume only three possible values—$k-1, k, k+1$—whenever process i tests $n.j$ following its phase k, $k > 0$, we may replace

$$n.j < k \quad \text{by} \quad n.j \bmod 3 = (k-1) \bmod 3$$

Similarly, the other tests involving $n.j$ can be replaced. We introduce variables $m.j$, where $m.j = n.j \bmod 3$. The tests within each loop can be written using $m.j$s. Now the variables $n.j$s can be eliminated because they are auxiliary variables (they do not appear in tests or assignments to other variables).

2.7 The Complete Algorithm

The protocol followed by process i is as follows.

- Initially, set every $m.j$ to 0 (in arbitrary order).
- Upon completion of phase 1 execute
  $s.i$:= set of all process indices;
  do $\langle [\!] \; j \;\; ::$
  $\quad j \in s.i \;\wedge\; m.j = 0 \rightarrow m.i \;:=\; 1$
  $\quad [\!] \; j \in s.i \;\wedge\; m.j = 1 \rightarrow s.i, m.i \;:=\; s.i - \{j\}, 1$
  $\quad [\!] \; j \in s.i \;\wedge\; m.j = 2 \rightarrow s.i \;:=\; \phi$
  $\rangle$
  od

- Upon completion of the phase k, $k > 1$, execute
  $s.i$:= set of all process indices; $m.i \;:=\; k \bmod 3$;
  do $\langle [\!] \; j \;\; ::$
  $\quad j \in s.i \;\wedge\; m.j = (k-1) \bmod 3 \rightarrow$ skip
  $\quad [\!] \; j \in s.i \;\wedge\; m.j = k \bmod 3 \quad \rightarrow s.i \;:=\; s.i - \{j\}$
  $\quad [\!] \; j \in s.i \;\wedge\; m.j = (k+1) \bmod 3 \rightarrow s.i \;:=\; \phi$
  $\rangle$
  od

Acknowledgment: I am grateful to J. R. Rao for a thorough reading of the manuscript. The idea of each processor setting all (relevant) variables to 0 initially and then setting its own variables to nonzero values subsequently also appears in [3] in connection with assigning identities to processors.

References

1. Chandy, K. M., and L. Lamport [1985]. Distributed Snapshots: Determining Global States of Distributed Systems, *ACM TOCS*, **3**:1, February 1985, 63–75.

2. Chandy, K. M., and J. Misra [1988]. *Parallel Program Design: A Foundation*, Reading, Massachusetts: Addison-Wesley, 1988.

3. Lipton, R. M. and A. Park [1988]. The Processor Identity Problem, *Information Processing Letters*, **36**:2, October 1990.