Phase Synchronization

Notes on UNITY: 12-90

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

1/31/90

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 which guarantees that:

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

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 $(k+1)^{th}$ phase 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 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 each of which read values from a common store, compute and write to 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

*This work was partially supported by ONR Contract 26-0679-4200 and by Texas Advanced Research Program grant 003658–065.

phase thus preventing the remaining processes from starting their next phase. It is easy to take global snapshots [1985] or system 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 the 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 the regular computation while some process is still taking a checkpoint.

Phase synchronization can be solved by using a general synchronization algorithm; see Chapter 14 of [1988]. In this paper we show a very simple scheme for solving this problem.

2 An Informal Description of the Solution

In this section, 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 its k^{th}phase, and a process enters its $(k+1)^{th}$ phase 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 k^{th}phase, $k \geq 1$, process i checks if all $n.j$ are at least k; if such is the case it begins its $(k+1)^{th}$ phase; otherwise, it sets $n.i$ to k.

Protocol for process i

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

The test, $\langle \exists\ j \quad :: \quad n.j < k \rangle$, 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 $(k+1)^{th}$ phase, $k \geq 0$, until all processes complete their k^{th}phase, and
- (Progress) If all processes have completed their k^{th}phase, $k \geq 0$, each process will enter its $(k+1)^{th}$ phase.

Since all processes have completed their 0^{th} phase 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 $(k+1)^{th}$ phase until process j completes its k^{th}phase. Process i sets $n.j$ to 0 initially and it detects $n.j \geq k$ before entering its $(k+1)^{th}$ phase. 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 k^{th} phase. Therefore, process j must have completed its k^{th}phase for the condition $n.j \geq k$ to hold.

2.2.2 Progress

Suppose that all processes have completed their k^{th}phase, $k > 0$. We show that every process will begin its $(k+1)^{th}$ phase eventually. When all processes have completed their k^{th}phase either (1) $\langle \forall\ j \quad :: \quad n.j \geq k \rangle$ holds, or (2) $\langle \exists\ j \quad :: \quad n.j < k \rangle$ 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 writes into $n.i$ after completing its k^{th}phase, $k > 0$, and (2) process i writes k or a higher value into $n.i$ after completing its k^{th}phase. Therefore, eventually $\langle \forall\ j \quad :: \quad n.j \geq k \rangle$ holds and continues to hold. Hence, every process will exit its loop (after the k^{th}phase) and begin the $(k+1)^{th}$ phase.

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 $(k+1)^{th}$ phase. We introduce a local variable, $s.i$, of process i that is used to store 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 k^{th}phase, $k > 0$, is as follows.

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

Note on notation: We use "$[\!] \ j \quad ::$" 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.

2.3.2 Proof of Synchronization

We show that at any point after process i assigns to $s.i$ upon completing its k^{th}phase and before it enters its $(k+1)^{th}$ phase, all processes outside $s.i$ have completed their k^{th}phase. The assignment to $s.i$ of all process indices, after process i completes its k^{th} phase, 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 k^{th}phase. Thus, if process i detects $n.j \geq k$ for some j in $s.i$, then process j has completed its k^{th}phase. Process i enters its $(k+1)^{th}$ phase 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 their k^{th}phase by then.

2.3.3 Proof of Progress

After all processes complete their k^{th}phase 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, $\langle \forall\ j \quad :: \quad n.j \geq k \rangle$ will hold eventually and continue to hold thereafter. Therefore every process will eventually complete its loop.

Note: These informal, operational proofs should not be taken too seriously (see a formal proof in Section 4). Such proofs can be employed to prove many incorrect programs as well. The reader may consider the following variation of the solution where the assignment

$s.i, n.i \ := \ s.i - \{j\}, k$
is replaced by
$s.i \ := \ s.i - \{j\}$

Is the solution correct? Do the proofs still go through? □

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

The variable $n.i$ need be set only once following completion of the k^{th}phase by process i, for $k > 1$ (but not $k = 1$). To see this, note that when process i completes its k^{th}phase all processes have completed their $(k-1)^{th}$ phase; 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 k^{th}phase, $k > 1$, 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 0^{th} phase (i.e., during their initializations).

2.5 A Shortcut in Testing

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

- upon completion of the first phase 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 k^{th}phase, $k > 1$, 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

Note: The alert reader should wonder why $n.i$ is not being set to 1 along with the assignment $s.i \ := \ \phi$, in the first case. □

2.6 Computing Modulo 3

The given solution requires 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.

We note that whenever process i tests $n.j$ after completing its k^{th}phase, $k > 0$,

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

To see the upper bound, since process i has not begun its $(k+1)^{th}$ phase—and hence, it has not completed its $(k+1)^{th}$ phase—process j has not begun (nor completed) its $(k+2)^{th}$ phase; therefore, $n.j \leq k + 1$. Also, when process k started its k^{th}phase, $k > 1$, 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.

As $n.j$ can assume only three possible values—$k - 1, k, k + 1$—whenever process i tests $n.j$ following its k^{th}phase, $k > 0$, 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 the first phase execute
  $s.i$:= set of all process indices;
  do $\langle [\!] \; j \;\; :: \;\; j \in s.i \;\wedge\; m.j = 0 \rightarrow m.i \;:=\; 1 \;\; [\!] \; j \in s.i \;\wedge\; m.j = 1 \rightarrow s.i, m.i \;:=\; s.i - \{j\}, 1 \;\; [\!] \; j \in s.i \;\wedge\; m.j = 2 \rightarrow s.i \;:=\; \phi \rangle$
  od

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

3 A Formal Description of the Protocol

Operational description of the protocol, as given so far, appeals to many programmers because it directly prescribes the code for each process. Unfortunately, such descriptions require "operational" proofs, i.e., reasoning about unfolding sequences of computations—a truly error prone scheme. In

this section we show how to describe a version of the protocol and prove its correctness, formally. We employ UNITY [2].

There is no program counter in UNITY; therefore we employ variables to keep track of the phases begun or completed by processes. Let $b.i$ be the last phase begun by process i and $e.i$ be the last phase ended by process i. (There is no notion of a process in UNITY. Thus, $b.i, e.i$ are simply program variables that are manipulated in prescribed ways—e.g., $e.i$ is set to $b.i$ to simulate ending of a phase by process i.) Initially, all $b.i, e.i$ are assumed to be 0, signifying that each process has (begun and) ended its 0^{th} phase.

As before, we have the variables $s.i, n.i$ for each process i. In program PS below, we encode an abstract version of the protocol of Section 2.3.1; some of the differences from that protocol are as follows.

- Variable $s.i$ is assigned (to the set of all processes) when a phase is begun; previously, this assignment was made upon completion of a phase. The difference is trivial.
- Program PS is more general in the sense that $n.i$ is assigned the number of the last completed phase—$e.i$—at any time during computation. Previously, this assignment took place only after completing a phase and before starting the next phase.
- Program PS is less general in the sense that initialization by a process is now carried out as an atomic step. This is only for convenience; a later refinement could allow for nonatomic initialization.
- In program PS each phase terminates because $e.i$ is eventually set to $b.i$, for every i. This assumption does not affect the correctness of the protocol.

Program PS {Phase Synchronization}
{The programming notation is from Chapter 2 of [2]}

initially $\langle [\!] \; i \;\; :: \;\; b.i, e.i = 0, 0 \rangle$

assign
$\langle [\!] \; i \;\; :: \;\;$ {the code for process i}

{process i initializes and begins its phase 1. Here, P is the set of all processes}
$\langle \| \; j \;\; :: \;\; b.i, s.i, n.j \;:=\; 1, P, 0 \;\; \text{if} \;\; b.i = 0 \rangle$

$[\!]$ {set $n.i$ to the number of the last completed phase, any time}
$n.i \;:=\; e.i$

$[\!]$ {end a phase}
$e.i \;:=\; b.i$

$[\!]$ {begin next phase if $s.i = \phi \;\wedge\; b.i \neq 0$}
$b.i, s.i \;:=\; b.i + 1, P \;\; \text{if} \;\; s.i = \phi \;\wedge\; b.i \neq 0$

$[\!]$ {remove an index j from $s.i$ if $b.i \leq n.j \;\wedge\; b.i \neq 0$}
$\langle [\!] \; j \;\; :: \;\; s.i \;:=\; s.i - \{j\} \;\; \text{if} \;\; b.i \leq n.j \;\wedge\; b.i \neq 0 \rangle$
$\rangle$

end

4 A Formal Proof of Correctness

Our proof obligations are (see Chapter 3 of [2] for logical notations employed here):

- (Synchronization)

  $\langle \forall\ i, j \quad :: \quad b.i \leq e.j + 1 \rangle$

- (Progress)

  $\langle \forall\ i \quad :: \quad \langle \forall\ j \quad :: \quad e.j \geq k \rangle \ \mapsto \ b.i > k \rangle$

4.1 Proof of Synchronization

We will prove the invariants, I1 and I2, given below. The invariants are proven by showing that initially they hold and each statement execution preserves their truths. We first prove invariant I1; then, we prove I2 using I1 in its proof.

$$\text{I1} \quad :: \quad \langle \forall\ i \quad :: \quad 0 \leq e.i \leq b.i \rangle$$

$$\begin{array}{lll} \text{I2} & :: & \langle \forall\ i \quad :: \quad b.i \neq 0 \ \Rightarrow \\ & & \qquad \langle \forall\ j \quad :: \quad n.j \leq e.j \ \wedge \ b.i \leq e.j + 1 \ \wedge \ (j \in s.i \ \vee \ b.i \leq e.j) \rangle \\ & & \rangle \end{array}$$

Predicate I1 is an invariant because it holds initially—all $e.i, b.i$ are 0—and execution of each statement preserves its truth: For any i, setting $e.i$ to $b.i$, or increasing $b.i$ by 1 preserves I1, and the remaining statements do not modify the variables named in I1. To prove the invariance of I2 note that I2 holds initially because every $b.i = 0$. Next, consider execution of each statement in turn.

Consider the "initialization" statement for some u. The postcondition, I2, may be written using two conjuncts for $i = u$ and $i \neq u$.

$$\begin{array}{l} [b.u \neq 0 \ \Rightarrow \ \langle \forall\ j \quad :: \ n.j \leq e.j \ \wedge \ b.u \leq e.j + 1 \ \wedge \ (j \in s.u \ \vee b.u \leq e.j) \rangle] \\ \wedge \ \langle \forall\ i \ : \ i \neq u \quad :: \quad b.i \neq 0 \ \Rightarrow \ \langle \forall\ j \quad :: \quad n.j \leq e.j \ \wedge \ b.i \leq e.j + 1 \ \wedge \ (j \in s.i \ \vee \ b.i \leq e.j) \rangle \rangle \end{array}$$

Using the axiom of assignment we substitute $1, P, 0$ for $b.u, s.u$ and every $n.j$, to obtain the precondition to be proven

$$\begin{array}{l} 1 \neq 0 \ \Rightarrow \ \langle \forall\ j \quad :: \quad 0 \leq e.j \ \wedge \ 1 \leq e.j + 1 \ \wedge \ (j \in P \ \vee \ 1 \leq e.j) \rangle \\ \wedge \ \langle \forall\ i \ : \ i \neq u \quad :: \quad b.i \neq 0 \ \Rightarrow \ \langle \forall\ j \quad :: \quad 0 \leq e.j \ \wedge \ b.i \leq e.j + 1 \ \wedge \ (j \in s.i \ \vee \ b.i \leq e.j) \rangle. \end{array}$$

Using I1, we may conclude that $0 \leq e.j$ and $1 \leq e.j + 1$. Also, $j \in P$ for all j. Hence, the required precondition follows from I2.

Execution of $n.i := e.i$, for some i, preserves $n.i \leq e.i$, and hence preserves I2. Execution of $e.i := b.i$, for some i, has the effect of—using I1—either increasing $e.i$ or leaving it unchanged; in either case I2 is preserved. The statement that assigns to $b.i$ and $s.i$ is executed under the precondition $s.i = \phi \ \wedge \ b.i \neq 0$; hence we conclude from I2 that the precondition $\langle \forall\ j \quad :: \quad n.j \leq e.j \ \wedge \ b.i \leq e.j \rangle$ holds prior to execution of this statement. Since the execution of the statement increases $b.i$ by 1, $\langle \forall\ j \quad :: \quad b.i \leq e.j + 1 \rangle$ holds as a postcondition, and, since $s.i$ is set to P, $\langle \forall\ j \quad :: \quad j \in s.i \rangle$ holds as a postcondition; thus, I2 holds as a postcondition. The statement that removes j from $s.i$ is carried out under the precondition $b.i \leq n.j$ and $b.i \neq 0$. Using I2 as a precondition, we conclude

$$b.i \neq 0 \ \Rightarrow \ n.j \leq e.j \ \Rightarrow \ \{\text{using } b.i \leq n.j\} \ \ b.i \leq e.j$$

Hence $b.i \leq e.j$ holds as a postcondition, thus preserving I2.

The proof of synchronization—that for all $i, j,\ b.i \leq e.j + 1$—follows by considering two cases: $b.i = 0$ and $b.i \neq 0$.

$$b.i = 0 \ \Rightarrow\ b.i \leq 1 \ \Rightarrow\ \{e.j \geq 0 \text{ from I1}\}\ \ b.i \leq e.j + 1$$
$$b.i \neq 0 \ \Rightarrow\ \{\text{from I2}\}\ \ b.i \leq e.j + 1.$$

4.2 Proof of Progress

We claim that the following properties hold for program PS. (In each property "i" is a free variable signifying that the property holds for all i. In property A5, variable "A" is free, signifying that A5 holds for all A.)

A1. $b.i \geq 0\ \ \textit{ensures}\ \ b.i > 0$

A2. $\langle \forall\ k\ :\ k \geq 0\ \ ::\ \ e.i \geq k\ \ \textit{ensures}\ \ e.i \geq k\ \wedge\ n.i \geq k \rangle$

A3. $\langle \forall\ k\ :\ k \geq 0\ \ ::\ \ e.i \geq k\ \wedge\ n.i \geq k\ \ \text{stable} \rangle$

A4. $\langle \forall\ k\ :\ k > 0\ \ ::\ \ b.i \geq k\ \wedge\ s.i = \phi\ \ \textit{ensures}\ \ b.i > k \rangle$

A5. $\langle \forall\ k\ :\ k > 0\ \ ::$
$\quad b.i \geq k\ \wedge\ \langle \forall\ j\ \ ::\ \ n.j \geq k \rangle\ \wedge\ s.i = A\ \wedge\ s.i \neq \phi$
$\quad\quad\quad \textit{ensures}$
$\quad b.i \geq k\ \wedge\ \langle \forall\ j\ \ ::\ \ n.j \geq k \rangle\ \wedge\ s.i \subset A$
$\rangle$

A property of the form p *ensures* q is proven for a program by showing that (1) for each statement s in the program, $\{p \wedge \neg q\}\ \ s\ \ \{p \vee q\}$ holds, and (2) there is a statement t in the program for which $\{p \wedge \neg q\}\ \ t\ \ \{q\}$ holds; the statement t "establishes" the property. We note that A1 is established by the initialization statement for i, A2 by the statement $n.i\ :=\ e.i$, A4 by the statement that assigns to $b.i$ and $s.i$, and A5 by the statement that removes an element from $s.i$. We leave the proof of stability in A3 and the remaining proofs that each property is "suitably preserved" by each statement, to the reader.

From A1–A5, we derive for every i and $k > 0$

A6. $b.i \geq k\ \wedge\ \langle \forall\ j\ \ ::\ \ n.j \geq k \rangle\ \mapsto\ b.i > k$

Proof: We consider two cases: $s.i = \phi$ and $s.i \neq \phi$. Applying disjunction to A6.1, A6.2, we get A6.

A6.1) $s.i = \phi$: $b.i \geq k\ \wedge\ \langle \forall\ j\ \ ::\ \ n.j \geq k \rangle\ \wedge\ s.i = \phi\ \mapsto\ b.i > k$

$$\begin{array}{ll} & b.i \geq k\ \wedge\ \langle \forall\ j\ \ ::\ \ n.j \geq k \rangle\ \wedge\ s.i = \phi \\ \mapsto & \{\text{A4, using the definition of } \mapsto\} \\ & b.i > k \end{array}$$

A6.2) $s.i \neq \phi$: $b.i \geq k\ \wedge\ \langle \forall\ j\ \ ::\ \ n.j \geq k \rangle\ \wedge\ s.i \neq \phi\ \mapsto\ b.i > k$

$$\begin{array}{ll} & b.i \geq k\ \wedge\ \langle \forall\ j\ \ ::\ \ n.j \geq k \rangle\ \wedge\ s.i \neq \phi\ \wedge\ s.i = A \\ \mapsto & \{\text{A5, using the definition of } \mapsto\} \\ & b.i \geq k\ \wedge\ \langle \forall\ j\ \ ::\ \ n.j \geq k \rangle\ \wedge\ s.i \subset A \\ \Rightarrow & \{\text{predicate calculus}\} \\ & (b.i \geq k\ \wedge\ \langle \forall\ j\ \ ::\ \ n.j \geq k \rangle\ \wedge\ s.i \neq \phi\ \wedge\ s.i \subset A) \\ \vee & (b.i \geq k\ \wedge\ \langle \forall\ j\ \ ::\ \ n.j \geq k \rangle\ \wedge\ s.i = \phi) \end{array}$$

Using the induction rule—finite sets, $s.i$, are well founded under subset ordering—

$$\begin{array}{l} b.i \geq k\ \wedge\ \langle \forall\ j\ \ ::\ \ n.j \geq k \rangle\ \wedge\ s.i \neq \phi\ \ \mapsto \\ b.i \geq k\ \wedge\ \langle \forall\ j\ \ ::\ \ n.j \geq k \rangle\ \wedge\ s.i = \phi \end{array}$$

Applying transitivity with A6.1, we get

$$b.i \geq k \ \wedge \ \langle \forall \ j \ :: \ n.j \geq k \rangle \ \wedge \ s.i \neq \phi \ \mapsto \ b.i > k \qquad \Box$$

Now we prove the progress property, A7, for every i and $k \geq 0$.

A7. (Progress) $\langle \forall \ j \ :: \ e.j \geq k \rangle \ \mapsto \ b.i > k.$

A7.1) $k = 0$:

$\langle \forall \ j \ :: \ e.j \geq 0 \rangle$

$\Rightarrow$ $\{e.i \leq b.i$ from I1$\}$

$b.i \geq 0$

$\mapsto$ {A1, using the definition of $\mapsto$}

$b.i > 0$

A7.2) $k > 0$: For any j and $k > 0$,

$e.j \geq k \ \mapsto \ e.j \geq k \ \wedge \ n.j \geq k$

, A2 and using the definition of $\mapsto$

$e.j \geq k \ \wedge \ n.j \geq k$ stable

, A3

$\langle \forall \ j \ :: \ e.j \geq k \rangle \ \mapsto \ \langle \forall \ j \ :: \ e.j \geq k \ \wedge \ n.j \geq k \rangle$

, completion rule on the above two

Rewriting the consequence

$\langle \forall \ j \ :: \ e.j \geq k \rangle \ \wedge \ \langle \forall \ j \ :: \ n.j \geq k \rangle$

$\Rightarrow$ $\{\langle \forall \ j \ :: \ e.j \geq k \rangle \ \Rightarrow \ e.i \geq k$, for a specific $i\}$

$e.i \geq k \ \wedge \ \langle \forall \ j \ :: \ n.j \geq k \rangle$

$\Rightarrow$ {from I1, $b.i \geq e.i$}

$b.i \geq k \ \wedge \ \langle \forall \ j \ :: \ n.j \geq k \rangle$

$\mapsto$ {from A6}

$b.i > k$

Thus, $\langle \forall \ j \ :: \ e.j \geq k \rangle \ \mapsto \ b.i > k \qquad \Box$

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, pp. 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," *TR # CSE–88–16*, Division of Computer Science, University of California, Davis, CA 95616.