Wait-free Consensus in "In-phase" Multiprocessor Systems*

Marina Papatriantafilou

Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
& CTI & Patras University, Greece
ptrianta@mpi-sb.mpg.de

Philippas Tsigas

Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
tsigas@mpi-sb.mpg.de

Abstract

In the consensus *problem in a system with n processes, each process starts with a private input value and has to choose irrevocably a decision value, which was the input value of some process of the system; moreover, all processes have to decide on the same value. This work deals with the problem of* wait-free*—fully resilient to processor crash and napping failures—consensus of n processes in an "in-phase" multiprocessor system. It proves the existence of a solution to the problem in this system by presenting a protocol which ensures that each process will reach decision within at most $n(n-3)/2+3$ steps of its own in the worst case, or within n steps if no process fails.*

1 Introduction

In the consensus problem in a system with n processes, each process starts with a private input value and has to choose irrevocably a decision value, which should be *valid*, i.e. it should equal the input value of some process, and *consistent*, i.e. it should be the same value for all processes. Whereas this is no problem in an ideal, failure-free environment, it imposes certain constraints on the capabilities of an actual system, which is viable only if it permits protocols tolerant to failures. In a system with failures the consensus problem becomes a central issue of multiprocessor synchronization and coordination. Solutions which guarantee that each process decides after a certain number of its own steps, regardless of the other processes' relative speeds, are called *wait-free*. Wait-freedom is a desirable property in concurrent systems, since it helps in taking advantage of the inherent parallelism in the system by ensuring that no process may be blocked by others which might be slow, preempted, swapped out, delayed without warning by interrupts; moreover, it implies maximum tolerance to processor crash and napping failures.

As expected, such a fundamental problem received much attention; as a result, many faces of the problem have been studied. In [12] and [22] it has been proven that in completely asynchronous systems–message passing and shared memory, respectively–not even one processor crash can be tolerated by a deterministic consensus protocol. In [9] the result is generalized for message passing systems; several critical parameters are identified and it is examined how they affect the number of faults that can be tolerated by a consensus protocol. In [14] shared memory data objects are partially classified according to the number of processes that can reach consensus in a wait-free manner using them. Recently, three groups with [7], [17] and [25], concurrently and independently have proven a conjecture first stated in [8], that even in the case when the agreement condition is weakened so that the decision values produced may differ, there is no protocol to tolerate k failures, where k is the maximum number of distinct values that may be chosen as decisions. Of particular interest was the introduction of algebraic and combinatorial topology in the study of these problems ([17, 25, 16, 18]). On the other hand, since in the asynchronous model the fault-tolerant consensus problem cannot be solved deterministically, solutions that have been given employ

*Partially supported by the EC ESPRIT II BRA ALCOM II (contr. #7141).

randomization or assume some form of synchrony. For an introduction and more references cf. [11], [23].

If we want to sum these up, from the theoretical point of view we have many surprising negative results, while the interest in the problem remains high. This is because, on one hand, it is interesting to develop a thorough understanding of the borders and relations between classes of objects with respect to their synchronization power; on the other hand, it is interesting to study more up-to-date architectures, which provide more fundamental synchronization primitives than just atomic reads and writes. Besides, it is easily noticeable that there is an important middle ground between the completely asynchronous and the completely synchronous extremes; this middle ground is reasonable for modeling real concurrent systems. As a result, there is an increasing interest in the past few years in research towards defining and designing new architectures (e.g. the transactional memory [15]) or towards exploiting the properties of already existing ones (those properties which were not all present in the theoretical models), in order to implement wait-free shared data objects ([13, 2, 6], to mention but a few).

1.1 Results and comparison with previous work

Following the direction mentioned above, in this work we consider the wait-free consensus problem in an "in-phase" multiprocessor system. In this system processors share a common clock pulse; in the duration of a pulse a processor reads the shared data of *one* processor, does some local computation and updates *its own* shared data. (It should be pointed out that a processor cannot modify the contents of registers owned by other processors.) It is possible that processes in this system operate at very different speeds because of pulse misses due to preemption, interrupts, page faults, or even processor crashes. So, although in a step a process atomically reads and writes, and, therefore, 2-process wait-free consensus is solvable ([14]), due to pulse misses and possible processor crashes, it is not obvious whether the n-process consensus problem can be solved wait-free in the system.

In the same system model the wait-free clock synchronization problem has been earlier studied in [10, 24]. Since by today's technology multiprocessor computers have large numbers of processors and since the probability of a crash increases with the number of processors in the system, it is vitally important both to study which multiprocessor can support protocols which tolerate faults, as well as to design such fault-tolerant protocols for them.

This work presents a solution for the wait-free consensus problem with n processes in an "in-phase" multiprocessor system, thus answering an open question stated in [10] and showing that this system model/architecture is strong enough to support deterministic n process fault tolerant agreement. The protocol ensures that a process will reach decision within $n(n-3)/2+3$ steps of its own in the worst case, or within $n-1$ steps when no process misses a pulse.

To the best of our knowledge no solution to this problem has been given before. Previous results that could *serve* as solutions can be found in [1, 4, 5]. Those protocols are for the the asynchronous model, which supports only read or write atomically, and guarantee decision in exponential, $O(n^4)$ and $O(nlog^2n)$ expected number of steps, respectively, but at the cost of randomization. On the other hand, the n process wait-free consensus protocols presented in [14] require some form of multi-writer read-modify-write or more sophisticated primitives (augmented queue, memory-to-memory-swap); the system model studied here does not provide that directly. Besides, the three protocols presented in [9] as part of a thorough analysis of the cases when consensus is solvable in message-passing systems, cannot be translated into solutions for our system model. Protocols E2 and E3 (of [9]) assume synchronous processes (no napping faults) and totally ordered messages (not just FIFO channels), respectively. Protocol E1 relies much on the nature of synchronous message communication, i.e. that a process which had a long napping failure receives all the messages sent to it during that time interval as soon as it resumes execution. In our model it can happen that a process has to make $n-1$ steps to learn about shared variable modifications; during that time it might suffer a new napping fault and this might be repeated unbounded many times.

2 The Computation Model

The system consists of n processes which are identified by distinct identity numbers, denoted by $P_1, \ldots, P_n$. The processes communicate via a set of single-writer, multi-reader atomic registers. Each one owns a subset of these registers. The owner of a register can write the register while all the other processes can read it. A *step* by a process P_k consists of the following actions:
(i) read by P_k of the shared registers owned by some process $P_{k'}$ $(k \neq k')$,
(ii) transition of P_k's local state (program counter, local variables), and
(iii) update of its own shared registers.

We consider "in-phase" multiprocessor systems, in which all processors share a common clock pulse. Each pulse is a (possibly empty) set of process names; the set of processes that *make a step* in the pulse. Each process can make at most one step in one pulse; if it does not make a step in some pulse it will be said to *miss* that pulse. A *configuration* is a tuple of process states and values of the shared variables. A system *execution* is a sequence $c_0\pi_1c_1\pi_2\ldots$ of alternating pulses (denoted by π_x) and configurations (denoted by c_x); consecutive pulses are indexed with consecutive integers. Each configuration c_i in a system execution is derived from its directly preceding configuration c_{i-1} by the state transitions and the shared variable updates of the processes that make a step in pulse π_i; the reads of shared registers that occur in pulse π_i return the respective values of c_{i-1}, while the updates of the shared registers in the same pulse take place in unison to derive c_i.

This system can be viewed as modeling either a PRAM (cf. [20, 21]) with faults or a multiprocessor synchronous system (cf. [19]) in which scheduling of the processes in different processors is done independently. Pause intervals can be interpreted as periods during which some process is not scheduled in a processor or is delayed without warning by interrupts, preemption, faults in the connections of the pausing processor or even processor crashes.

For any pulse π_i in any system execution E and for any process P_k, let $work(P_k, \pi_i)$ denote the number of steps that P_k made from the beginning of E (pulse π_1) until (and including) pulse π_i.

In the consensus problem each process P_k is given an input value v_k and is required to return an output value v'_k; we call the step when this happens the *return step* of P_k. If P_k makes its return step in some pulse π_i, it makes no more steps in subsequent pulses in the execution; if at some pulse in an execution P_k has not made its return step yet, it is called *undecided* in that pulse. A *wait-free consensus* protocol should satisfy the following *requirements* for every system execution:

Wait-freedom There should be a bounded number T such that: there is no process P_k and pulse π_i such that P_k is undecided in π_i and $work(P_k, \pi_i) > T$.

Validity For each process P_k, its output value v'_k should equal the input value $v_{k'}$ of some process $P_{k'}$ of the system.

Consistency For any processes P_k and $P_{k'}$ with output values v'_k and $v'_{k'}$ it should be $v'_k = v'_{k'}$.

3 Description of the Protocol

The protocol is described in C-like pseudocode in figures 1 and 2. We have adopted several conventions, like using capital case names for shared variables and capital boldface for calls to read/write shared registers. The following paragraphs describe the protocol more intuitively.

Each process P_k $(k \neq 1)$ first plays a game with P_1. If it wins it is called *dominant* in the set $\{P_1, ..., P_k\}$ and writes that information on its DOM_k shared register (we also write "dominant" as a shorthand for "dominant in the set $\{P_1, ..., P_k\}$"). Subsequently, the final decision is reached through stages in which partial decisions are made inductively. Consider an arbitrary execution E and let $\mathcal{D}_{1..proc}$ denote the decision that would be taken in the execution E' derived from E by considering only processes $P_1, \ldots, P_{proc}$ (i.e. if the steps of the other processes in E are eliminated). The protocol tries to follow the inductive rule:

$$\mathcal{D}_{1..n} = \begin{cases} VAL_n, & \text{if } P_n \text{ is dominant} \\ \mathcal{D}_{1..(n-1)}, & \text{otherwise} \end{cases}$$

$$\mathcal{D}_{1..1} = \quad VAL_1$$

where VAL_k, equals the input value of P_k. Each process tries to find $\mathcal{D}_{1..proc}$ for $proc = 1, \ldots, n$. After a process P_k makes a partial decision for $\mathcal{D}_{1..proc}$, it writes the value v decided and the process identity

```
var (VAL_1, DOM_1, D_SET_1, DEC_1), ..., (VAL_n, DOM_n, D_SET_n, DEC_n): (vtype, boolean, 1..n, vtype) ;
                                        /* Shared var's; Initially all = (null, 0, 1, null) */

DECIDE(k, val)                                              /* process P_k with input value val */
  var proc, d_set, d_set_p : 1..n ;                                        /* Initially = 1, 1, 1 */
      dom, dom_p, rech: boolean ;                                          /* Initially = 0, 0, 0 */
      dec, dec_p, val_p: vtype ;                                           /* Initially arbitrary */
                                                  /* Local variables are globally binded in the module */

  proc READ&check(i)          /* Read P_i's registers as first action in a step; check for decisions */
  begin
      READ (val_p, dom_p, d_set_p, dec_p) from (VAL_i, DOM_i, D_SET_i, DEC_i) ;
      if d_set_p ≥ i then d_set := d_set_p; dec := dec_p endif                /* copy advanced decision */
  end

  proc UPDATE()                                          /* Update own var's as last action of step */
  begin
      WRITE (val, dom, d_set, dec) to (VAL_k, DOM_k, D_SET_k, DEC_k) ;
  end

  proc recheck(l : 1..n, dec_tmp: vtype)
  var i : 1..n ;
  begin
      for (i := 2; i < l; i++) do
        if i ≠ k do
          READ&check(i) ;
          if d_set ≥ l then proc := i := d_set endif                        /* advance current process ptr */
          if (i = l − 1 or i = l − 2 and k = l − 1) and d_set < l then d_set := l; dec := dec_tmp endif
          UPDATE() ;
        endif
      endfor
  end
```

Figure 1: Protocol DECIDE: (a) Shared Variables and auxiliary procedures

proc on its DEC_k and D_SET_k shared registers, respectively. Let $D_k(proc) = v$ denote that mapping, which is the estimation of P_k for $\mathcal{D}_{1..proc}$; P_k will finally return its $D_k(n)$. Since processes might miss pulses and are, therefore, asynchronous, deviations from the rule for deciding $\mathcal{D}_{1..proc}$ are allowed, so that a fast process $P_{k'}$ can meet the wait-free requirement when P_1 and P_{proc} are so slow that the result of the game between them is not known at the time that $P_{k'}$ needs to find out $\mathcal{D}_{1..proc}$. The deviation is that the fast process $P_{k'}$ (arbitrarily) considers that P_{proc} is not dominant and sets $D_k(proc) = D_k(proc - 1)$. Since this is done by a fast process, i.e. early enough, that information is available in the respective shared registers, for the slow processes to find out about the deviation from the rule, and, therefore, decide consistently.

```
proc SafePhase(k)                        /* Estimate D_{1..k}= decision if system consited only of P_1..P_k */
begin
  READ&check(proc) ;                     /* step S_k(1): dominance in {P_1,...,P_k}? (check presence of P_1) */
  if d_set > 1 then proc := d_set else if val_p = null then dom = 1 endif endif
  if dom and k = 2 then d_set := 2; dec := val endif
  UPDATE() ;
  for (++proc; proc < k; proc++) do
    READ&check(proc) ;
    if d_set ≥ proc then proc := d_set                          /* advance current process ptr */
    else if ¬(dom) then d_set := proc ;      /* if dom don't care; interesetd only in advanced decisions */
            if dom_p then dec := val_p endif endif               /* else dec = D_k(proc) = D_k(proc − 1) */
    endif ;
    if proc = k − 1 and d_set < k then d_set := proc := k ; if dom then dec := val endif endif ;
    UPDATE() ;
  endfor
end

begin                                                            /* Main body of procedure DECIDE */
  if k = 1 then dec := val endif                                 /* Step S_k(0): announce presence */
  UPDATE() ;
  if k ≠ 1 then SafePhase(k) endif
  for (++proc; proc ≤ n; proc++) do
    READ&check(proc) ;
    if d_set ≥ proc then proc := d_set ;                         /* advance current process ptr */
    else if dom_p then
            if k = 1 then d_set = proc; dec := val_p
            else dec_tmp := val_p; rech := 1 endif          /* should recheck for "deviate" decisions */
    else d_set = proc endif                       /* consider proc not dominant ("deviate" decision) */
    endif
    UPDATE() ;
    if rech then recheck(proc, dec_tmp); rech := 0 endif
  endfor
  return(dec) ;
end
```

Figure 2: Protocol DECIDE: (b) Procedure $SafePhase(k)$ and main body of DECIDE

Naturally, each process, in each one of its steps, checks whether a final decision or a more *advanced*—than the one it knows so far—partial decision is reached and adopts it, thus advancing its process scanning pointer, i.e local variable *proc*.

The game between P_1 and an arbitrary P_k is played as follows: Each process (including P_1), as its first action of the protocol—*announcement* or 0-step—simply announces its participation in the game by writing its input value on its VAL_k register. In its next step, each P_k ($k \neq 1$) reads the register of P_1 and becomes dominant only if it reads that P_1 has not made its announcement step yet (i.e. if $VAL_1 = null$); in all other cases P_k looses the game. P_1 plays with each P_{proc} in $\{P_2, \dots, P_n\}$, (excluding only advances of its local variable *proc*) by checking whether P_{proc} is dominant; if not, P_{proc} will never become dominant, since it will read that P_1 has already made its announcement step. Thus, in that case the partial decision is $\mathcal{D}_{1..proc} = \mathcal{D}_{1..(proc-1)}$ and P_1 sets $D_1(proc) = D_1(proc-1)$. Otherwise—if P_{proc} is dominant—, P_1 safely (for reasons that become clear later in this section) concludes that $\mathcal{D}_{1..proc} = VAL_{proc}$ and sets $D_1(proc) = VAL_{proc}$.

Each process P_k ($k \neq 1$) follows a protocol of 2 stages; first, using procedure $SafePhase(k)$ it estimates the partial decision $\mathcal{D}_{1..k}$ and then continues for $\mathcal{D}_{1..(k+1)}, \dots, \mathcal{D}_{1..n}$. After making an observation, we describe the second phase first, to give the intuition in a more clear way.

Observation 1 *If a process P_k is not dominant and reads $DOM_{k'} = 0$, then it knows that $P_{k'}$ will not become dominant in that execution, because it knows that P_1 has made its announcement step.*

Suppose that P_k has an estimation (from procedure $SafePhase(k)$) of the partial decision $\mathcal{D}_{1..k}$. Then, for each $proc = k+1, \dots, n$, it estimates $\mathcal{D}_{1..proc}$ (excluding advances of its local variable *proc*), as follows: if it reads $DOM_{proc} = 0$, i.e. that P_{proc} is not dominant, then, only if P_k itself is not dominant, it can safely make a conclusion, namely that P_{proc} will never become dominant (by observation 1); otherwise, it cannot conclude in a wait-free manner (i.e. without waiting for P_1 and/or P_{proc} to make a step) anything about this future. Therefore, it arbitrarily considers P_{proc} not dominant and writes that $D_k(proc) = D_k(proc-1)$ (by leaving its DEC_k shared register unchanged and updating only its D_SET_k into *proc*). This implies that any other process $P_{k'}$ ($k' < proc$), which will possibly later read that P_{proc} is dominant, before feeling free to decide $D_{k'}(proc) = VAL_{proc}$, it will have to *recheck* for earlier, deviate decisions about $\mathcal{D}_{1..proc}$, among processes with identity P_x, s.t. $x < proc$ (i.e. among processes that might have had to decide about that with insufficient information). Note that P_1 need not recheck for deviations in decisions for any $\mathcal{D}_{1..proc}$ if it reads that P_{proc} is dominant; if such a decision was made, this would have happened before the first step of P_{proc}, therefore before the 0-step of P_1 itself, by a process P_x, s.t. $x < proc$. Thus, P_1 would have read that decision before reading P_{proc}'s registers.

Now, the description of the first phase: in procedure $SafePhase(k)$, if P_k becomes dominant, in which case it should be $\mathcal{D}_{1..k} = VAL_k$, it only has to check whether a fast process in $\{P_2, \dots, P_{k-1}\}$ has earlier made a deviate decision about $\mathcal{D}_{1..k}$; if P_k is not dominant it has to estimate $\mathcal{D}_{1..(k-1)}$. In both cases it suffices for P_k to scan once the shared registers of each one of the processes in $\{P_2, \dots, P_{k-1}\}$. For the case that P_k is *not* dominant, if it reads $DOM_{proc} = 0$ for some P_{proc}, then it knows that P_{proc} will never become dominant (by observation 1) and decides $D_k(proc) = D_k(proc-1)$; otherwise, if it sees that P_{proc} is dominant, P_k decides $D_k(proc) = VAL_{proc}$, without having to *recheck* for earlier, deviate decisions for $\mathcal{D}_{1..proc}$, because if there was any, P_k would have read it before reading P_{proc}'s registers (by an argument similar to the one in the previous paragraph).

4 Correctness and Performance of the Protocol

For what follows in this section, we introduce some auxiliary notation/terminology. Consider an arbitrary system execution E:

- A process P_k maps a value v to the set $\{1, \dots, proc\}$ in E if there exists a configuration such that $DEC_k = v$ and $D\_SET_k = proc$; we denote this mapping by $D_k(proc)$ and say that P_k *decides* this value v for the set $\{1, \dots, proc\}$. This might happen either because P_k *copies* that decision from the shared register of some other process (in *READ&check*) or because P_k *computes* that decision (in $SafePhase$ on in the main body of $DECIDE$). Note that P_k in its return step in E returns $D_k(n)$. In analogy with consistency for the final output value, we say that the decisions for a set $\{1, \dots, proc\}$ are *consistently* made in E by the processes of a set $\mathcal{P}$, if for any two $P_k, P_{k'} \in \mathcal{P}$ that decide for $\{1, \dots, proc\}$, it holds that $D_k(proc) = D_{k'}(proc)$.

- If s and s' are steps by processes, $s \rightarrow s'$ denotes that s precedes s', while $s \,\|\!\rightarrow s'$ denotes that s either precedes or is concurrent with s' in E; the latter is equivalent with $\neg(s' \rightarrow s)$. The step of P_k in which it reads P_{proc}'s shared registers for the first time—if ever—and not during procedure *recheck*, is denoted by $S_k(proc)$. Furthermore, $S_k(0)$ denotes the announcement, or 0-step of P_k.

- A process P_k is *dominant* in E, if in the configuration after $S_k(1)$—and, henceforth in all subsequent configurations in E— it is $DOM_k = 1$.

Lemma 1 (Wait-freedom) *In a system with n processes, in any execution, each process P_k makes its return step after having made at most $n(n-3)/2+3$ steps.*

Proof Each process P_k decides $D_k(k)$ in at most k steps of its own; for each $D_k(x)$ $(k < x \leq n)$ it needs in the worst case—i.e. if *recheck* is necessary— $(x-2)$ steps of its own. Summing this all up, we have that in the worst case P_k terminates using $DECIDE$ in

$$k + \sum_{x=k+1}^{n} (x-2) = \frac{n(n-3)}{2} - \frac{k(k-5)}{2}$$

steps of its own. Since P_1 never *recheck*s and, hence, terminates in at most n steps, the largest value of the above expression is for $k=2$ and equals $n(n-3)/2+3$ steps. □

Lemma 2 (Validity) *In each execution each process which makes a return step outputs a value that equals the input value of some process in the execution.*

Proof *(Sketch)* A process P_k that terminates returns the value $D_k(n)$ that decides and holds in its DEC_k shared variable. That value came either from a copy from some process $P_{k'}$'s $DEC_{k'}$ shared variable, or from an assignment to DEC_k of the input value VAL_{proc} of a process P_{proc}. In the latter case the lemma is straightforward; in the former case, if we trace back the origin of the value held in $DEC_{k'}$, by the same argument, we will find that it is an input value of some process in that execution. □

Lemma 3 (Consistency) *In a system with n processes the decisions for all sets $\{1,\ldots,p\}$ $(1 \leq p \leq n)$ are consistently made by the processes of the set $\{P_1,\ldots,P_n\}$ in every system execution.*

Proof *(Sketch)* This can be proven by induction on p.

Consider an arbitrary execution E in a system with n processes. For $p=1$ the lemma holds because for any process P_{proc} in $\{P_1,\ldots,P_n\}$, in order to decide $D_{proc}(1)$ it should be $S_{proc}(1) \rightarrow S_1(0)$, which implies that the input value of P_1 is available in its VAL_1 shared variable; therefore, for each P_{proc} that decides for $\{1,\ldots,1\}$, it will be $D_{proc}(1) = VAL_1$.

Assume the lemma holds for all $p \leq k$; we will show that it also holds for $p = k+1$. From the induction hypothesis we know that the decisions for $\{1,\ldots,k\}$ are consistently made by the processes of the set $\{P_1,\ldots,P_n\}$. In order to prove that the decisions for $\{1,\ldots,k+1\}$ are also consistently made by the processes of the set $\{P_1,\ldots,P_n\}$, there are two cases to be considered:

Case 1: P_{k+1} is *not dominant* in E. Then each P_{proc} in $\{P_1,\ldots,P_k\} \cup \{P_{k+2},\ldots,P_n\}$ that makes $S_{proc}(k+1)$ in E, reads $DOM_{k+1} = 0$ and, therefore, can only possibly set $D_{proc}(k+1) = D_{proc}(k)$. Besides, P_{k+1} itself can only possibly set $D_{k+1}(k+1) = D_{k+1}(k)$. Since the decisions for $\{1,\ldots,k\}$ are consistently made by the processes in $\{P_1,\ldots,P_n\}$, the same follows in this case for the decisions for $\{1,\ldots,k+1\}$.

Case 2: P_{k+1} is *dominant* in E. There are two possibilities:

- Each P_{proc} in $\{P_1,\ldots,P_k\} \cup \{P_{k+2},\ldots,P_n\}$ that executes $S_{proc}(k+1)$ and decides for $\{1,\ldots,k+1\}$ reads $DOM_{k+1} = 1$ (which implies that there are no "deviate" decisions), and, therefore, sets $D_{proc}(k+1) = VAL_{k+1}$; similarly, P_{k+1} can only possibly set $D_{k+1}(k+1) = VAL_{k+1}$ (in the end of procedure *SafePhase*). Therefore, in this case the decisions for $\{1,\ldots,k,k+1\}$ are consistently made by the processes in $\{P_1,\ldots,P_n\}$.

- Some P_{proc} in $\{P_1,\ldots,P_k\}$, which executes $S_{proc}(k+1)$ and decides for $\{1,\ldots,k+1\}$ reads $DOM_{k+1} = 0$. Clearly, $proc \neq 1$ and $proc \neq k+1$; otherwise, the dominance of P_{k+1} would be contradicted. P_{proc} computes $D_{proc}(k+1) = D_{proc}(k)$. The following are true in E:

$$S_{proc}(k+1) \,\|\!\rightarrow S_{k+1}(1) \,\|\!\rightarrow S_1(0)$$

since P_{k+1} is dominant in E, but P_{proc} reads $DOM_{k+1} = 0$;

$$S_{k+1}(1) \to S_{k+1}(proc) \,\|\!\!\to S_{k+1}(k)$$

since $1 < proc \leq k$ and P_{k+1} makes those steps in that order;

$$S_1(0) \to S_1(proc) \to S_1(k+1)$$

again because $0 < proc < k+1$ and P_1 makes those steps in that order. Since in $S_{proc}(k+1)$ process P_{proc} computes $D_{proc}(k+1) = D_{proc}(k)$, the above precedence relations imply that P_1 and P_{k+1} will copy that decision (or a more "advanced" than that one, of course) from P_{proc}'s shared register in their $S_1(proc)$ and $S_{k+1}(proc)$ steps, respectively, i.e. before the steps in which they would have to decide for $\{1, \ldots, k, k+1\}$ ($S_1(k+1)$ and $S_{k+1}(k)$, respectively). Therefore, $\{P_1, P_{proc}, P_{k+1}\}$ decide consistently.

Now, for any other $P_{proc'}$ in $\{P_1, \ldots, P_k\}$ ($proc' \neq 1, proc, k+1$) that executes $S_{proc'}(k+1)$ and decides for $\{1, \ldots, k+1\}$, we have either that $S_{proc'}(k+1) \,\|\!\!\to S_{k+1}(1)$ or that $S_{k+1}(1) \to S_{proc'}(k+1)$. In the former case $P_{proc'}$ reads $DOM_{k+1} = 0$ and sets $D_{proc'}(k+1) = D_{proc'}(k)$, therefore, we are done. In the latter case $P_{proc'}$ reads $DOM_{k+1} = 1$ and executes *recheck*. Since P_{proc}, $P_{proc'}$ read from DOM_{k+1} values $0, 1$, respectively, it holds that

$$S_{proc}(k+1) \,\|\!\!\to S_{k+1}(1) \to S_{proc'}(k+1)$$

Moreover, from the protocol we have that

$$S_{proc'}(k+1) \to \{recheck(k+1, VAL_{k+1}) \; by \; P_{proc'}\}$$

Since P_{proc} in $S_{proc}(k+1)$ computes $D_{proc}(k+1) = D_{proc}(k)$, the above precedence relations imply that $P_{proc'}$ will copy that decision from P_{proc}'s shared register during *recheck*. Therefore, we are done with all the processes in $\{P_1, \ldots, P_{k+1}\}$.

Now, for any $P_{proc''}$ in $\{P_{k+2}, \ldots, P_n\}$ that executes $S_{proc''}(k+1)$ and decides for $\{1, \ldots, k+1\}$, we have the following: (a) If $P_{proc''}$ is dominant in E, since $k+1 < proc''$, from the protocol it follows that $P_{proc''}$ can only copy $D_{proc''}(k+1)$ from a process in $\{P_1, \ldots, P_k\}$ (it is not interested in computing it), therefore, we are done. (b) If $P_{proc''}$ is not dominant in E and reads $DOM_{k+1} = 0$, it will decide $D_{proc''}(k+1) = D_{proc''}(k)$, which is known by the induction hypothesis that it is consistently decided; if it reads $DOM_{k+1} = 1$ then the following are true in E:

$$S_{proc}(k+1) \,\|\!\!\to S_{k+1}(1) \,\|\!\!\to S_1(0)$$

because P_{k+1} is dominant, but P_{proc} reads $DOM_{x+1} = 0$;

$$S_1(0) \to S_{proc''}(1)$$

since $P_{proc''}$ is not dominant in E;

$$S_{proc''}(1) \to S_{proc''}(proc) \to S_{proc''}(k+1)$$

because $1 < proc < k+1$ and $P_{proc''}$ makes those steps in that order. Since P_{proc} in $S_{proc}(k+1)$ computes $D_{proc}(k+1) = D_{proc}(k)$, the above precedence relations imply that in its $S_{proc''}(proc)$ step, $P_{proc''}$, will copy that decision into its $DEC_{proc''}$. Since the decisions for $\{1, \ldots, k\}$ are known to be consistently made by the processes in $\{P_1, \ldots, P_n\}$, the same follows in this case for the decisions for $\{1, \ldots, k+1\}$.

□

Considering the requirements from a solution to the wait-free consensus problem, the previous lemmas imply the following theorem:

Theorem 1 *The* **DECIDE** *protocol correctly implements a wait-free solution to the consensus problem in an in-phase multiprocessor system with n processes, with $T = n(n-3)/2 + 2$.*

Conclusions

In this paper it is shown that the in-phase multiprocessor system model is strong enough to support wait-free—i.e. tolerating processor crash and napping failures—solutions to the n-process consensus problem. Previously it has been shown that the same architecture can support wait-free and self-stabilizing clock synchronization protocols. Both the consensus protocol and the clock synchronization protocols have quadratic time complexity; since both problems are central issues in fault-tolerant multiprocessor coordination, it is interesting to either prove that this is also the lower bound or to find more efficient solutions.

References

[1] K. Abrahamson On Achieving Consensus Using a Shared Memory. *Proceedings of 7th Annual ACM Symposium on Principles of Distributed Computing (PODC 1988)*, pp. 291-302.

[2] J. Alemany, E. W. Felten Performance Issues in Non-Blocking Synchronization on Shared Memory Multiprocessors. *Proceedings of 11th Annual ACM Symposium on Principles of Distributed Computing (PODC 1992)*, pp. 125–134.

[3] J. Aspnes, M. Herlihy Wait-free data structures in the Asynchronous PRAM. *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 1990)*, 340-349.

[4] J. Aspnes, M. Herlihy Fast Randomized Consensus Using Shared Memory. *Journal of Algorithms* **11**, pp. 441-461 (1990).

[5] J. Aspnes, O. Warts Randomized Consensus in Expected $O(n \log^2 n)$ Operations Per Processor. *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 1992)*, pp. 137-146.

[6] G. Barnes A Method for Implementing Lock-Free Shared Data Structures. *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 1993)*, pp. 261–270.

[7] E. Borowsky, E. Gafni Generalized FLP Impossibility Results for t-resilient Asynchronous Computations. *Proceedings of the 25th Annual ACM Symposium on the Theory of Computing (STOC 1993)*, pp. 91-100.

[8] S. Chaudhuri Agreement Is Harder Than Consensus: Set Consensus Problem in Totally Asynchronous System Systems. *Proceedings of 9th Annual ACM Symposium on Principles of Distributed Computing (PODC 1990)*, pp. 311-324.

[9] D. Dolev, C. Dwork, L. Stockmeyer On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, vol. 34, No. 1, January 1987, pp. 77-97.

[10] S. Dolev, J.L. Welch Wait-free clock synchronization. *Proceedings of 12th Annual ACM Symposium on Principles of Distributed Computing (PODC 1993)*, pp. 97-108.

[11] M.J. Fischer The Consensus Problem in Unreliable Distributed Systems (A Brief Survey). *YALEU/DCS/RR-273*, June 1983.

[12] M. Fischer, N. Lynch, M. Paterson Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, Vol. 32, No. 2, April 1985, pp. 374-382.

[13] M. Herlihy A Methodology for Implementing Highly Concurrent Data Objects. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 1990)*, pp. 197–206.

[14] M. Herlihy Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 1, January 1991, pp. 124-149.

[15] M. Herlihy, J.E.B. Moss Transactional Memory: Architectural Support for Lock-Free Data Structures. *Proceedings of the 20th Annual International Symposium on Computer Architectures (ISCA 1993).*

[16] M. Herlihy, S. Rajsbaum Set Consensus Using Arbitrary Objects. *Proceedings of 13th Annual ACM Symposium on Principles of Distributed Computing (PODC 1994)*, pp. 324-333.

[17] M. Herlihy, N. Shavit The Asynchronous Computability Theorem for t-resilent Tasks. *Proceedings of the 25th Annual ACM Symposium on the Theory of Computing (STOC 1993)*, pp. 111-120.

[18] M. Herlihy, N. Shavit The Asynchronous Computability Theorem for Wait-free Computation. *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing (STOC 1994).*

[19] K. Hwang. Advanced Computer Architectures, Parallelism, Scalability, Programmability. McGraw-Hill, Inc. 1993.

[20] R. Karp, V. Ramachandran. Parallel Algorithms for Shared Memory Machines. J.van Leeuwen, ed., *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity* Elsevier, Amsterdam 1990.

[21] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, Inc., 1992.

[22] M.C. Loui, H. H. Abu-Amara Memory Requirements for Agreement among Unreliable Asynchronous Processes. *Advances in Computing Research*, Vol. 4 (1987), pp. 163-183.

[23] N.A. Lynch, I. Saias Distributed Algorithms: Lecture Notes *MIT/LCS/RSS 16 Research Seminar Series 1993.*

[24] M. Papatriantafilou, Ph. Tsigas Wait-free Self-Stabilizing Clock Synchronization. *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory (SWAT 1994)*, Springer-Verlag Lecture Notes in Computer Science Vol. 824, pp. 267–277.

[25] M. Saks, F. Zaharoglou Wait-Free k-set Agreement is Impossible: The Topology of Public Knowledge. *Proceedings of the 25th Annual ACM Symposium on the Theory of Computing (STOC 1993)*, pp. 101-110.