The Las-Vegas Processor Identity Problem (How and When to Be Unique)*

Shay Kutten
Department of Industrial Engineering
The Technion
kutten@ie.technion.ac.il

Rafail Ostrovsky
Bellcore
rafail@bellcore.com

Boaz Patt-Shamir
Department of Electrical Engineering–Systems
Tel-Aviv University
boaz@eng.tau.ac.il

March 29, 1998

Abstract

We study the problem of assigning unique identifiers to identical concurrent processes. The model considered in this paper is the asynchronous shared memory model, and the correctness requirement is that upon termination of the algorithm, the processes must have unique IDs *always*. We give tight positive and negative results for the problem. Our main positive result is the first Las-Vegas protocol that solves the problem. The protocol terminates in $O(\log n)$ expected time, using $O(n)$ shared memory space, where n is the number of participating processes. The new protocol improves on all previous solutions simultaneously in running time (exponentially), probability of termination (to 1), and space requirement. On the negative side, we show that there is no finite-state Las-Vegas protocol for the model where schedules may depend on the history of the shared variables. We also show that any Las-Vegas protocol must know n in advance, and that the running time is $\Omega(\log n)$. For the case of an arbitrary (non-oblivious) adversary, we present a Las-Vegas protocol that uses $O(n)$ unbounded registers. For the read-modify-write model, we present a deterministic algorithm which uses $O(1)$ shared bits.

1 Introduction

In the course of designing a distributed algorithm, one has often to confront the problem of how to distinguish among the different participating processes. This problem, called the *Processor Identity Problem* (abbreviated PIP), has applications in some of the most basic distributed tasks, including mutual exclusion, choice coordination, and resource allocation.

One attractive (and sometimes realistic) solution is simply to deny the existence of the problem

*An Extended Abstract of this paper appeared in the Proceedings of the Second Israel Symposium on Theory of Computing Systems, June 1993.

altogether: just assume that each process is generated with a unique identifier. In some cases, however, one cannot get away with this approach [9]. In this paper we study the following variant of PIP, first suggested by Lipton and Park. We are given a shared-memory asynchronous system of n processes. Each process has a designated private output register, which is capable of storing $N \geq n$ different values. All shared registers can be written by all processes and read by all processes. A protocol for PIP must satisfy the following conditions.

Symmetry: The processes execute identical programs (in particular, they have identical initial state).

Uniqueness: Upon termination, all the values in the output registers are distinct.

In this paper, we study the problem in the *read-write registers* model (also known as *atomic registers* [7]), where in a single action, a processor can either read or write the contents of a single register and perform some local (possibly probabilistic) computation. It is easy to see that in this model no deterministic protocol can solve PIP. For randomized protocols, we need a refined correctness requirement. If one is willing to tolerate erroneous termination (i.e., termination of the protocol with some ID shared by two or more processes), then there exists a trivial solution that does not require any communication: each process chooses independently a random ID; the error probability is controlled by the size of the ID space. In this paper we restrict our attention to the case where erroneous termination is absolutely disallowed.

We distinguish between two types of algorithms for PIP. A *Monte-Carlo* algorithm for PIP is one which guarantees with high probability that the algorithm terminates. In the case of failure, the protocol never terminates (but if it terminates, then the IDs are guaranteed to be unique). A *Las-Vegas* algorithm for PIP is one which has finite expected time for termination (again, where termination is such that the uniqueness is guaranteed). Note that a Las-Vegas algorithm must have probability 1 for termination.

Previous Work. Symmetry breaking is one of the well-studied problems in the theory of distributed systems. For example, see [2, 6, 3, 1]. The Processor Identity Problem where errors are forbidden was first defined by Lipton and Park in [9]. Their formulation (motivated by a real-life system) contains two additional requirements. First, they require that the algorithm would work regardless of the initial state of the shared memory; in other words, the initial contents of the shared registers is arbitrary (the "dirty memory" model). Secondly, they require that upon termination, the set of IDs output by the algorithm is exactly the set $\{1, \ldots, n\}$. In [9], Lipton and Park give a Monte-Carlo solution that for any given integer $L \geq 0$ uses $O(Ln^2)$ shared bits and terminates in $O(Ln^2)$ time with probability $1 - c^L$, for some constant $c < 1$. The protocol in [9] is Monte-Carlo, i.e., failure results in a non-terminating execution. In [11], Teng gives an improved Monte-Carlo algorithm for PIP. Teng's algorithm uses $O(n \log^2 n)$ shared bits, and guarantees, with probability $1 - 1/n^c$ (for a constant $c > 0$), running time of $O(n \log^2 n)$. Both algorithms [9, 11] have the interesting property that their outcome depends only on the coin-flips made by the processes, and it is independent of the order in which the processes take steps.

Concurrently with our work, Eğecioğlu and Singh [4] have obtained a Las-Vegas algorithm for PIP that works in $O(n^7)$ expected time using $O(n^4)$ shared bits.

Our Results and Their Interpretation. In this work, we present tight positive and negative results describing the conditions under which the Las-Vegas Processor Identity Problem can be solved. On the positive side, we give the first Las-Vegas algorithms for PIP in the read-write registers model. This protocol improves on all previous protocols simultaneously in termination probability, running time, and shared space requirement. Specifically, the protocol terminates in (optimal) $O(\log n)$ expected time, using shared space of size $O(n)$ bits.[1] As in the original formulation of PIP, the protocol has the nice features that it works in the dirty memory model, and that upon termination the given names constitute exactly the set $\{1, \ldots, n\}$. Our algorithm is developed in a sequence of refinement steps, introducing a few techniques which may be of interest in their own right. For example, in the course of developing the protocol, we obtain, as a by-product, a simple and efficient protocol for the *dynamic* model, where processes may join and leave the system dynamically. The dynamic version of PIP is relaxed to require that if no process joins the system for sufficiently long time, then eventually the IDs stabilize on unique values.

The correctness of our algorithms relies crucially on a few assumptions. First, we assume that the execution has an *oblivious schedule*, i.e., the order in which processes take steps does not depend on the history of the unfolding execution. Secondly, for the terminating (non-dynamic) algorithm, we require that n, the number of participating processes, is known in advance and can be used by the protocol.

Our negative results show that the assumptions we made are actually necessary to obtain a Las-Vegas protocol for PIP, even if the shared memory is initialized. First, we prove that any Las-Vegas protocol that solves PIP must know n precisely in advance, even if the schedule is oblivious. Our most interesting negative result concerns the nature of the schedules. We show that even if n is known, there is no finite-state Las-Vegas protocol for PIP which works for *adaptive schedules*, i.e., in the case where an adversary can decide which process moves next, based on the history of the shared variables during the execution. This impossibility result is complemented with a Las-Vegas protocol for PIP which works for any (fair) schedule—using unbounded space. We suspect that the technique developed to prove this lower bound may be applicable for other distributed Las-Vegas problems.

We also prove lower bounds on the complexity of algorithms for PIP. We show that any Las-Vegas algorithm for PIP works in time $\Omega(\log n)$. We complete the picture by considering a much stronger computational model, where in a single indivisible step a process can read a shared register and update its value. For this model, we show that there exists a *deterministic* protocol for PIP which uses a single shared variable with $O(1)$ different values.

There are several ways to interpret our results. For instance, we now understand why the algorithms of [9, 11] have positive probability of non-termination: this follows from the fact that these algorithms are indifferent to the schedule. In particular, the algorithms will have the same outcome even if the schedule is adaptive, and hence, by our impossibility result, they must have positive probability of non-termination. Another interpretation of our results is that in the Las-Vegas model, one can have either a bounded space protocol, or a protocol which is resilient to arbitrary adversaries, but not both. Other interpretations regard the fault-resilience of protocols for PIP: the fact that n must be known

[1] In our algorithm, some registers are of size $O(\log n)$ bits, whereas the algorithms of [9, 11, 4] uses only one bit registers.

in advance means that there is no terminating algorithm for PIP which can tolerate even a single crash-failure. Our complexity results imply that our algorithm is optimal in terms of time. It can also be shown that the time-space product of any algorithm for PIP satisfies $TS = \Omega(n)$ [10], which implies that our algorithm is nearly optimal in terms of space. We leave open the question whether a sublinear-space algorithm exists.

Organization of This Paper. We start, in Section 2, with a description of the models we consider. In Section 3, we develop algorithms for PIP in the read-write model. In Section 4 we derive impossibility results. Finally, in Section 5, we give an algorithm for PIP in the read-modify-write model.

2 The Asynchronous Read-Write Shared Memory Model

Our basic model is the asynchronous shared-memory distributed system. Below, we give a definition of the formal underlying system: in specifying algorithms, we shall use a higher-level pseudo-code, that is straightforward to translate into the formalism below.

We start by defining formally the notion of a system. We first give a general definition, and later refine the distributed nature of the system.

Definition 2.1 *A* system *is a set of m* registers $\{r_1, \ldots, r_m\}$ *and n* processes $p_1, \ldots, p_n$*. Each register r_j has an associated* domain*, which is a set of primitive elements called* values*. Each process p_i has an associated set of primitive elements called* local states*. A* global state *of the system is an assignment of values to registers and local states to processes. A* transition specification *of the system is a set of n probabilistic functions from global states to global states, where each function is associated with a distinct process.*

The distributed nature of a system is captured by placing appropriate restrictions on the transition specification. In this paper, we are mainly interested in the read-write shared memory model, where the transition specifications satisfy the following condition.[2]

Definition 2.2 *A system is a* read-write shared registers *system if for all transition specifications τ_i and all global states s, we have that in $s' = \tau_i(s)$, only p_i may change its local state, and at most one register may have changed its value, and one of the following holds.*

- *If all registers retain their values in s', then there exists a single register r_j such that the local state of p_i in s' is a probabilistic function of only the local state of p_i in s and the value assigned to r_j in s. We say that p_i* reads *r_j.*
- *If the values assigned to registers in s and s' differ for register r_j, then the value assigned to r_j is a probabilistic function of only the local state of p_i in s, and the local state of p_i in s' is a probabilistic function of only the local state of p_i in s. We say that p_i* writes *r_j.*

[2] In our model, the random choice is made *after* the memory access: given a state, the type of the next access of any process (either a read or a write) is determined. We remark that this modeling is done for convenience only and has no essential impact on our results.

A *terminating state* of process p_i is a state whose associated transition function for p_i is the identity function (deterministically).

Given the definition above, we model an *execution* of a system by a sequence of states, where in each step, one process applies its transition specification to yield the next state. To model asynchrony, we assume that the choice of which process takes the next step is under the control of an adversary. Two types of adversaries are considered in this work, defined as follows.

Definition 2.3 *Let $\mathcal{S}$ be a system with n processes. An* oblivious adversary *for $\mathcal{S}$ is an infinite sequence of elements of $\{1, \ldots, n\}$. An* adaptive adversary *for $\mathcal{S}$ is a function from the finite sequences of shared memory states to $\{1, \ldots, n\}$.*

Intuitively, an oblivious adversary commits itself to the order in which processes are scheduled to take steps oblivious to the actual execution, while an adaptive adversary observes the shared memory (which may depend on outcomes of coin-flips) and chooses which process to schedule next, based on the unfolding execution. In addition, we impose a general restriction on both types of adversaries, namely, they must be *fair*; this means, in our context, that each process must be scheduled to take steps infinitely often.

2.1 Time Complexity

To measure time in the asynchronous read-write model, we use the normalizing assumption that each process takes at least one step (either a read or a write) every one time unit. The executions are completely asynchronous, i.e., the processes have no access to clocks, and the notion of "time" is used only for the purpose of analysis. Using this time scale, we define the running time of an algorithm to be the time in which the last process reaches a terminating state, and infinity if some process never terminates.

3 A Solution for the Processor Identity Problem

In this section we present a protocol that solves the Processor Identity Problem in the asynchronous shared memory model. To aid in the exposition of the main ideas, we develop the algorithm in a few steps. First, we present a *dynamic* algorithm, i.e., an algorithm which works for a dynamically changing set of processes. The dynamic algorithm is non-terminating: we only show that the set of IDs assigned to processes *stabilizes* in $O(\log n)$ expected time units, using $O(n)$ shared bits, where n is the number of participating processes. Next, we augment the dynamic algorithm with a termination-detection mechanism; the resulting algorithm works in the *static* case, where the set of the participating processes is fixed, and their number, n, is known in advance. Adding termination detection incurs only a constant blowup in asymptotic complexity: the static algorithm halts in $O(\log n)$ expected time units, and uses $O(n)$ shared registers of $\log n$ bits each. Finally, we sketch the extensions which allow the final algorithm to solve the Processor Identity Problem in its original formulation [9], i.e., assuming the dirty memory model, and producing as the final names exactly the set $\{1, \ldots, n\}$.

Our terminating protocols work under the assumptions that n is known in advance, and that the schedule is oblivious. We shall show in Section 4 that both assumptions are necessary to ensure termination with probability 1.

3.1 A Dynamic Algorithm for PIP

We start our discussion by considering the *dynamic setting*, where processes may start and stop taking steps dynamically. More precisely, we assume that the execution is infinite, and that in each given point in the execution of the system, there is a set of *active processes*. While a process is active, it takes at least one step every time unit. Inactive processes do not take steps. In this model, the task of a PIP algorithm is to assign unique IDs to active processes, provided that the set of active processes does not change for a sufficiently long time interval. In this section, we present and analyze a dynamic algorithm for a system of n processes which uses $O(n)$ shared bits, and stabilizes in $O(\log m)$ time units, where m is the number of active processes.

<u>*Constants*</u>

n : *the total number of processes*
N : *$c \cdot n$ for some constant $c > 1$*

<u>*Shared Variables*</u>

$\mathcal{D}$: *a vector of N bits, initial value arbitrary*

<u>*Local Variables*</u>

old_sign : *in $\{0, 1, \perp\}$, initial value $\perp$*
ID : *output value in $\{0, \ldots, N-1\}$, initial value arbitrary*

<u>*Shorthand*</u>

random(S) : *returns a random element of S under the uniform distribution.*

<u>*Process Code*</u>

repeat forever
  either, with probability $1/2$, *do* — write a new signature and record its value
    $\mathcal{D}[ID] \leftarrow old\_sign \leftarrow random(\{0,1\})$
  or, with probability $1/2$, *do* — read signature and choose new ID if signature changed
    if $old\_sign \neq \mathcal{D}[ID]$ **then**
      $old\_sign \leftarrow \perp$ — no claim yet!
      $ID \leftarrow random(\{0, \ldots, N-1\})$
      $\mathcal{D}[ID] \leftarrow old\_sign \leftarrow random(\{0,1\})$ — make a claim for new ID

Figure 1: A dynamic algorithm for PIP.

The dynamic model here serves two goals. First, it may be of value on its own right, depending on the system requirement. Secondly, it allows us to isolate the problem of symmetry breaking from the difficulty of termination detection.

Pseudo-code for the dynamic algorithm is given in Figure 1. To facilitate the discussion, we make the following definitions.

Definition 3.1 *For a given global state:*

- *An active process p_i is said to* claim *an ID j if $ID_i = j$ and $old\_sign_i \neq \perp$.*
- *An ID j is called* unique *if it is claimed by exactly one process.*
- *A* collision *is said to occur at an ID j if j is claimed by more than one process.*

Note that by the code, a process claims an ID only after it has written $\mathcal{D}[ID]$.

Intuitively, the idea in the algorithm is as follows. While a process is active, it claims an ID, and repeatedly checks it to verify that it is not claimed by any other process. If a process detects a collision on its claimed ID, it "backs off" and chooses at random a new ID to claim.

The key property of the algorithm is that it guarantees, with a positive constant probability, that each collision is detected in $O(1)$ time units. To achieve this, the algorithm works as follows. Let n be the total number of processes in the system, i.e., the maximum number of processes which can be active concurrently. We use a a vector of N bit registers in shared memory, where $N = c \cdot n$ for some constant $c > 1$. Each register corresponds to a particular ID, namely the index of that register. In each iteration of the loop, a process p_i either reads the register corresponding to its claimed ID, or writes it. The choice between the alternatives is random. If p_i writes, it writes a random *signature*: a signature is just a bit, which is drawn independently at random whenever the process signs. The value of the last signature is recorded in the local memory of p_i. If p_i chooses to read, it examines the contents of the register to see whether it was changed since the last time p_i signed it. If a change is detected, then p_i concludes that another process claims the same ID p_i claims, or, in other words, that p_i's ID gives rise to a collision. In this case p_i (politely) chooses a new ID uniformly at random from $\{0, \ldots, N-1\}$, and then proceeds with this new ID to the next iteration. If no change was detected when p_i reads, it proceeds directly to the next iteration.

We now analyze the dynamic algorithm. Assume that there exists a suffix of the execution in which there is a fixed set of m active processes; we call this suffix the *static suffix*. In the following analysis, we restrict our attention to the static suffix only. The correctness of the algorithm is stated in the theorem below.

Theorem 3.1 *With probability $1 - m^{-\Omega(1)}$, $O(\log m)$ time units after the execution of the algorithm in Figure 1 starts and onwards, there are no collisions and no process changes its claimed ID. Moreover, the expected stabilization time is $O(\log m)$ as well.*

We first state and prove the following lemma, which is the basic probabilistic result we use.

Lemma 3.2 *There exist constants $0 < p, \mu < 1$ and a natural number K such that for any global state s, if $k \geq K$ processes claim non-unique IDs at s, then with probability at least $1 - p^k$, in $O(1)$ time units at least μk of these processes choose a new ID.*

Proof: Assume without loss of generality that processes $P = \{p_1, \ldots, p_k\}$ claim IDs $J = \{j_1, \ldots, j_l\}$ at state s, where each ID in J is claimed by at least two processes in P. Let σ denote the segment of the execution which starts at s and ends when all processes in P complete an iteration. Clearly, σ is $O(1)$ time units long. By the code, if a process claims an ID, it will access the corresponding entry in $\mathcal{D}$ in the following iteration. We now restrict our attention to a particular ID $j \in J$. Let $\sigma|_j = \left\langle a_1^j, a_2^j \ldots \right\rangle$ by the sequence of processor steps obtained from σ be leaving only the first access to $\mathcal{D}[j]$ by each process in P. For a given step a_i^j, let $p(a_i^j)$ be the process which takes step a_i^j. Let X_i^j be a random variable having value 1 if process $p(a_i^j)$ chooses a new ID at step a_i^j, and 0 otherwise (X_i^j is 0 also when a_i^j is undefined). First, we claim that

$$\Pr\left[X_i^j = 1 \mid i > 1\right] \;\geq\; \frac{1}{8} \tag{1}$$

To see this, consider the pair of steps a_{i-1}^j, a_i^j. Since $p(a_{i-1}^j) \neq p(a_i^j)$, the probability that $p(a_i^j)$ chooses a new ID at step a_i^j is at least the probability that (i) step a_{i-1}^j is a write of a value different than old_sign_i, and that (ii) step a_i^j is a read. By the code, (i) occurs with probability 1/4, and (ii) occurs with probability 1/2. By the assumption that the schedule is oblivious, we have that (i) and (ii) are independent, and Eq. (1) follows.

Next, let $\{Y_s\}_{s=1}^{\infty}$ be a sequence of independent identically distributed Bernoulli random variables with $\Pr[Y_s = 1] = \frac{1}{8}$. We claim that for any $\epsilon > 0$ we have that

$$\Pr\left[\sum_{j\in J, i\geq 1} X_{2i}^j < \epsilon\right] \;\leq\; \Pr\left[\sum_{s=1}^{k/3} Y_s < \epsilon\right] \tag{2}$$

To see this, consider the pairs of steps a_{2i}^j, a_{2i-1}^j for all i, j for which this pair of steps is defined. First we argue that there are at least $k/3$ such pairs: for each ID $j \in J$, let z^j be the number of processes which claim j; the number of (a_{2i}^j, a_{2i-1}^j) pairs is therefore $\lfloor \frac{z_j}{2} \rfloor \geq \frac{z_j}{3}$ since $z_j \geq 2$. Next, observe that the accesses in different pairs are disjoint, and hence independent. Eq. (2) now follows from Eq. (1).

Now, using the Chernoff bound we have that $\Pr\left[\sum_{s=1}^{k/3} Y_s < \frac{k}{32}\right] \leq e^{-\Theta(k)}$, and hence, by Eq. (2) we have that

$$\Pr\left[\sum_{j\in J, i\geq 1} X_{2i}^j < \frac{k}{32}\right] \;\leq\; e^{-\Theta(k)} \;.$$

The Lemma follows from the fact that the probability that less than $k/32$ of the k colliding processes choose a new ID is at least $\Pr\left[\sum_{j\in J, i\geq 1} X_{2i}^j < \frac{k}{32}\right]$. ∎

We also state the following simple facts.

Lemma 3.3 *Whenever a process chooses a new ID, this ID is unique with a positive probability independent of n.*

Proof: The number of claimed IDs is never more than n, and a new ID is chosen at random among $N = cn$ possibilities. ∎

Lemma 3.4 *If an ID is claimed at a given global state in an execution of the algorithm, it remains claimed throughout the execution.*

Proof: This follows from the definition which says that an ID is claimed only if its corresponding $\mathcal{D}$ entry was written by a process, and the observation that at all states of an execution, the last process to write the entry claims the corresponding ID. ∎

Using Lemmas 3.2 and 3.3, we prove Theorem 3.1.

Proof of Theorem 3.1: Consider the static suffix. Let D_K denote the event that at most K IDs are not unique. Combining Lemmas 3.2 and 3.3, we can deduce that there exists a constant $K > 0$ such that

$$\begin{aligned} \Pr[\text{in } O(\log m) \text{ time units, } \mathrm{D}_K \text{ occurs}] &\geq 1 - O(m^K \log m) \\ E[\text{time until } \mathrm{D}_K \text{ occurs}] &\leq O(\log m) \end{aligned} \tag{3}$$

To complete the proof, note that if in a given state, the number of non-unique IDs is bounded by a constant, then with some constant probability p_0 all processes will claim unique IDs in $O(1)$ time units, and hence

$$\begin{aligned} \Pr[\text{all processes claim unique IDs in } O(\log m) \text{ time} \mid D_K \text{ holds at start}] &\geq 1 - O(m^K \log m) \\ E[\text{time until all processes claim unique IDs} \mid D_K \text{ holds at start}] &\leq \frac{1}{p_0} \end{aligned} \tag{4}$$

Combining the bounds in (3,4) yields the desired result. ▮

3.2 A Las-Vegas Protocol for PIP

The next step in the development of the algorithm is to make the protocol terminating. To this end, we add the extra assumption that the number of participating processes is known in advance (thus disallowing process crashes). In this section we assume that all shared registers are initialized.

The termination detection mechanism is based on the observation that the set of claimed IDs is a non-decreasing function of time in each execution (Lemma 3.4). Therefore, it suffices to have a non-perfect mechanism to determine whether an ID is claimed: a mechanism which detects claimed IDs only eventually, so long as it never detects erroneously a non-claimed ID. This way, when a process detects n claimed IDs, it can safely deduce that the set of claimed IDs has stabilized on unique values.

Informally, this is done as follows. We embed a $\lceil \log N \rceil$-depth binary tree in the shared memory, where the leafs correspond to the set of N possible IDs. The idea is that each tree node v would contain the number of claimed IDs whose corresponding leafs are in the subtree rooted by v. To accomplish the intended meaning, each process repeatedly traverses the path in the tree which starts at the leaf corresponding to the current ID it claims, and ends at the root node, writing the sum of the children in each node it traverses. The criterion for exiting the loop is that the root holds the value n, which indicates that the set of claimed IDs has stabilized on unique values. Note that processes may overwrite each other's values: some extra care has to be exerted in order to avoid losing a value permanently.

For the tree nodes, we use the following notation.

Definition 3.2 *Let T be a complete binary tree of 2^h nodes. Let $0 \leq i \leq 2^h - 1$, and let $0 \leq \ell \leq h$. Then:*

- *$root_T$ denotes the root of T.*

- $ancestor_T(i, \ell)$ *denotes the level* ℓ *ancestor of the ith leaf.*
- $left_T(i, \ell)$ *denotes the left sibling of* $ancestor_T(i, \ell)$ *(and* $left_T(i, \ell) = ancestor_T(i, \ell)$ *if* $ancestor_T(i, \ell)$ *is a left child).*
- $right_T(i, \ell)$ *denotes the right sibling of* $ancestor_T(i, \ell)$ *(and* $right_T(i, \ell) = ancestor_T(i, \ell)$ *if* $ancestor_T(i, \ell)$ *is a right child).*

In Figure 2 we present pseudo-code for the termination detection algorithm alone. We have isolated *sum_childs* as a subroutine, and we shall reuse it later. The only connections of the termination detection algorithm of Figure 2 with the dynamic algorithm of Figure 1 is that the latter determines *ID*, and the former determines when to halt (when the number of claimed IDs is n).

<u>*Shared Variables*</u>
  T : *a complete binary tree of height* $\lceil \log N \rceil$*, initially all nodes are* 0

<u>*Local Variables*</u>
  ID : *claimed ID, controlled by the algorithm in Figure 1*

<u>*Shorthand*</u>
  $sum\_childs_S(i, \ell) \equiv$ read children of $ancestor_S(i, \ell)$ and write their sum in $ancestor_S(i, \ell)$
  1s **if** $\ell = 0$ **then** $ancestor_S(i, \ell) \leftarrow 1$
  2s **else**
  3s $count \leftarrow left_S(i, \ell - 1) + right_S(i, \ell - 1)$ — $count, count'$ are temporary variables
  4s $ancestor_S(i, \ell) \leftarrow count$
  5s $count' \leftarrow left_S(i, \ell - 1) + right_S(i, \ell - 1)$
  6s   **if** $count' > count$ **then**
  7s $count \leftarrow count'$
  8s   **go to** *4s*

<u>*Code*</u>
1 $\ell \leftarrow 0$
2 **while** $root_T < n$
3 $sum\_childs_T(ID, \ell)$
5 $\ell \leftarrow (\ell + 1) \bmod (\lceil \log N \rceil + 1)$
6 **halt**

Figure 2: Termination detection algorithm

We now state and prove the main properties of the algorithm in Figure 2. Since in this algorithm we have only a single tree, we omit the T subscript in the analysis. We start with a property of the *sum_childs* subroutine.

Lemma 3.5 *Let* $0 \leq i < N$*, and let* $0 \leq \ell < \lceil \log N \rceil$*. If in a given execution the values of* $left(i, \ell)$ *and* $right(i, \ell)$ *stop changing at time* t_0*, and if* $ancestor(i, \ell + 1)$ *is written at some time* $t_0 + O(1)$*, then the value of* $ancestor(i, \ell + 1)$ *stop changing at time* $t_0 + O(1)$*, and the stable values satisfy* $ancestor(i, \ell + 1) = left(i, \ell) + right(i, \ell)$*.*

Proof: Let w denote the first step in the execution after which both $left(i,\ell)$ and $right(i,\ell)$ stop changing. Clearly, w is a write step.

---

1. *Process q reads $left(i,\ell)$ and $right(i,\ell)$ before they stabilize.*
2. *Step w (time t_0): process p writes so that $left(i,\ell)$ and $right(i,\ell)$ become stable and correct.*
3. *Process q writes $ancestor(i,\ell+1)$ incorrectly (time t).*
4. *Process q reads $left(i,\ell)$ and $right(i,\ell)$.*
5. *Process q writes $ancestor(i,\ell+1)$ correctly.*

Figure 3: A summary of the scenario described in the proof of Lemma 3.5. There can be at most $O(1)$ time units between steps 2 and 5 above.

Call a write to a node *correct* if it writes the sum of the stable values of its two children. Consider any write to $ancestor(i,\ell+1)$ at time $t > t_0$. By the code, such a write is done by the subroutine *sum_childs*. Note that each write of line 4s to $ancestor(i,\ell+1)$ is preceded by reading $left(i,\ell)$ and $right(i,\ell)$ either in line 3s or in line 5s. Let d be the maximal time which can elapse between a read in lines 3s or 5s and the subsequent write of line 4s. Clearly, $d = O(1)$. Note that if $t - t_0 > d$, the value written by that step must be correct, since the values read aby the writing process must have been read after $left(i,\ell)$ and $right(i,\ell)$ have stabilized. So suppose that $t - t_0 < d$. We show that if the value written to $ancestor(i,\ell+1)$ at time t is incorrect, the correct value will be written in $O(1)$ additional time units. To see that, suppose that a process q writes $ancestor(i,\ell+1)$ incorrectly at time t (step 3 in Figure 3). Since the write is line 4s of the code, q subsequently reads $left(i,\ell)$ and $right(i,\ell)$ by line 5s (step 4 in Figure 3). Since these reads occur after the corresponding values have stabilized, the result stored in $count'$ is smaller than the value q has written into $ancestor(i,\ell+1)$, and hence q would go back to line 4s and write the correct value in $ancestor(i,\ell+1)$ (step 5 in Figure 3). Clearly, this write occurs in time $t + O(1) = t_0 + O(1)$. ∎

Next we prove a simple property of the summation process.

Lemma 3.6 *If at state s we have $ancestor(i,\ell) = v$, then in s, at least v IDs are claimed among the IDs corresponding to leafs in the subtree rooted at $ancestor(i,\ell)$.*

Proof: First, note that if $ancestor(i,\ell)$ is never written, then its contents is 0 by the initialization. So suppose that $ancestor(i,\ell)$ is written at some point. We prove the lemma in this case by induction on the levels. For $\ell = 0$ the lemma follows from Lemma 3.4, the definition of $sum\_childs(i,0)$ for a process claiming ID i, and the assumption that the leafs of the tree are initialized by the value 0. Consider now $\ell > 0$: when the contents of $ancestor(i,\ell)$ is written in line 3, it is the sum of $l = left(i,\ell-1)$ as read in a prior state s', and of $r = right(i,\ell-1)$ as read in another prior state s''. Since $left(i,\ell-1)$ and $right(i,\ell-1)$ are at level $\ell-1$, and since the number of claimed IDs in any set is non-decreasing by Lemma 3.4, it follows from the induction hypothesis that at state s there are at least $l+r$ claimed IDs in the subtree rooted by $ancestor(i,\ell)$. ∎

We can now summarize the properties of the algorithm.

Theorem 3.7 *There exists an algorithm for the Processor Identity Problem for n processes with expected running time $O(\log n)$, which requires $O(n)$ shared bits.*

Proof: The algorithm consists of running the algorithm of Figure 1 in parallel to the algorithm of Figure 2, using the following rule: after each memory access of the termination detection algorithm, it is suspended and a complete iteration of the algorithm of Figure 1 is executed; if the claimed ID has changed, then the termination detection algorithm is resumed at line 1 of Figure 2, and otherwise it is resumed where it was suspended. The combined algorithm halts when the termination detection algorithm halts.

Obviously, the only effect on the algorithm of Figure 1 is a constant slowdown and hence Theorem 3.1 holds for the combined algorithm as well. For the termination detection part, it is straightforward to verify that Lemma 3.6 holds in the combined algorithm. It follows from Theorem 3.1 that after $O(\log n)$ expected time units, all claimed IDs are unique, and hence all leafs of the summation tree have stabilized by Lemma 3.4. Noting also that by the code, $O(1)$ time units after a node $ancestor(i, \ell)$ have stabilized its parent $ancestor(i, \ell + 1)$ will be written (at least by the same process which stabilized $ancestor(i, \ell)$), we can apply Lemma 3.5 inductively, and prove that after $O(\log n)$ additional time units, the root of the summation tree contains n, and the algorithm halts.

To conclude the analysis of the algorithm, we bound the size of the shared space used. The collision detection part uses N bits. The termination detection algorithm uses $N/2^\ell$ registers of $2^\ell + 1$ states for each level $0 \le \ell \le \lceil \log N \rceil$ of the tree, and hence the total number of shared bits used in the summation is

$$\sum_{\ell=0}^{\log_2 N} \frac{(\ell+1)N}{2^\ell} < 4N \ .$$

Therefore, the overall number of shared bits required by the algorithm is less than $N + 4N = O(n)$. ∎

As an aside, we remark that one can use the summation tree constructed by the termination detection algorithm to determine, for each process, what is the *rank* of its final ID among all final IDs, thereby compacting the set of IDs to $\{1, \ldots, n\}$. This can be done in additional $O(\log n)$ time units using the prefix sums algorithm [8].

3.3 Las-Vegas protocol for PIP in the Dirty Memory Model

The algorithm as described in Theorem 3.7 works under the assumption that all memory entries are initialized. In this section we explain how to adapt the algorithm to work in the dirty memory model, where the initial contents of the shared memory is arbitrary.

One approach is to initialize the whole memory as a preliminary step. This approach (used in previous papers) takes a very long time: we have $\Omega(n)$ shared registers, and we cannot break the initialization task between the processes before we can distinguish among them. The approach we take, therefore, is to do "lazy" initialization: initialize registers only before they are used, in a way such that each process initializes only the portion required for its current state. This has to be done carefully: there is a danger of one process erasing irrecoverably the contents of a register which was written by a process which has already halted.

For the collision detection, note that the algorithm in Figure 1 works even if the shared memory is not initialized: this is true because the first access of a process to a shared register is always a write action. The termination detection algorithm, however, requires an initialized memory. We explain how to adapt the algorithm in Figure 2 to the dirty memory model.

<u>*Shared Variables*</u>

T_1, T_2 *:complete binary trees of height* $\lceil \log N \rceil$

<u>*Local Variables*</u>

ID *: claimed ID, controlled by the algorithm in Figure 1*

seen *: a Boolean flag, initially* FALSE — ever read n at $root_{T_1}$?

T_0 *: a complete binary tree of* $\lceil \log N \rceil$*, initially all* `dirty` — was $ancestor_{T_1}(i,\ell)$ written?

<u>*Additional Shorthand*</u>

$wipe(i,\ell) \equiv$ — initialize $ancestor_{T_1}(i,\ell)$ and children before doing $sum\_childs_{T_1}(i,\ell)$

  $ancestor_{T_0}(i,\ell) \leftarrow$ `clean`
  if $\ell > 0$ **and** $right_{T_0}(i,\ell-1) =$ `dirty` **then**
    $right_{T_0}(i,\ell-1) \leftarrow$ `clean`
    $right_{T_1}(i,\ell-1) \leftarrow 0$
  if $\ell > 0$ **and** $left_{T_0}(i,\ell-1) =$ `dirty` **then**
    $left_{T_0}(i,\ell-1) \leftarrow$ `clean`
    $left_{T_1}(i,\ell-1) \leftarrow 0$

<u>*Code*</u>

$\ell \leftarrow 0$
$root_{T_1} \leftarrow 0,\ root_{T_2} \leftarrow 0$
while $root_{T_2} < n$
  if $\neg seen$ **then**
    if $root_{T_1} = n$ **then**
      $seen \leftarrow$ TRUE
      $right_{T_2}(ID,\ell) \leftarrow 0,\ left_{T_2}(ID,h) \leftarrow 0$ *for all* $0 < h \leq \lceil \log N \rceil$
    $wipe(ID,\ell)$
  if *seen* **then**
    $sum\_childs_{T_2}(ID,\ell)$
  $sum\_childs_{T_1}(ID,\ell)$
  $\ell \leftarrow (\ell+1) \bmod (\lceil \log N \rceil + 1)$
halt

Figure 4: Code for termination detection with dirty shared memory

We modify the termination detection algorithm as follows. We maintain *two* trees, which we call the *first* and *second* tree (T_1 and T_2 in Figure 4). The first tree is used as in Section 3.2, with the following modification: each process maintains a local image of the first tree, in which it indicates whether each register is known to be "clean": initially, all registers are marked "dirty". Whenever the process writes to a register in the first tree, it marks the register "clean" in its local image.

Whenever a process is about to read a register from the first tree, it checks whether the register is marked "dirty"; if so, the process first writes 0 in that register and marks it "clean." Then the process proceeds to read that register. If the register is already marked "clean," then the process proceeds to read it immediately. The consequence of this modification is that no uninitialized register is ever read; however, some meaningful values may be overwritten, but never with values greater than their "correct" values. Since the processes keep re-writing their leaf-root paths in the first tree, it follows that $O(\log n)$ time units after all claimed IDs become unique, a value of n will be written at the root of the first tree. The problem now is that if a process will halt altogether, some values on its leaf-root path may be overwritten and never recovered.

To overcome this problem, we apply the tree-summation algorithm of Figure 2 to the second tree. However, this tree counts something different, and we use a different initialization strategy for it. Specifically, the second tree is used to count *how many processes have seen n at the root of the first tree*. When a process sees n at the root of the first tree, it starts working on the second tree using the same leaf, while keeping working on the first tree as usual. The point is that once any process have seen n at the root of the first tree, the set of claimed IDs has stabilized, and thus the leaf-root path of all processes will not change any more. Hence it suffices for each process to initialize only $O(\log n)$ registers in the second tree before it starts reading any value there, and never initialize any register later. This property implies that once a value of n is read at the root of the second tree, it will never be erased by any process, so that all processes will eventually halt.

The full code is given in Figure 4. In the following discussion, PIA denotes the algorithm presented in Sections 3.1 and 3.2, and PIB denotes the modified algorithm for the dirty memory model. We now analyze PIB. Call the steps in PIB which zero out dirty registers *pseudo initialization steps*.

One simple property of PIB is the following.

Lemma 3.8 *For any execution of PIB there exists an execution of PIA such that the values written by PIB in the first tree are at most the values written by PIA in the summation tree.*

Proof: Given an execution ρ of PIB, construct an execution of ρ' PIA by omitting all pseudo-initialization steps and keeping all the random choices. We claim, by induction on the length of ρ, that the values written and read in non pseudo-initialization steps of ρ are not greater than the corresponding values in ρ'. This can be seen by noting that any value written by the algorithm is the sum of two values read in the past, and noting that these values were written either in a non pseudo-initialization step where the induction applies, or otherwise it is 0. The claim follows, since all values ever written by PIA are non-negative. ▮

A direct corollary of Lemma 3.8 and Lemma 3.6 is following property of PIB.

Lemma 3.9 *If a process reads at state s $ancestor_{T_1}(i,\ell) = v$, then in s, at least v IDs are claimed among the IDs corresponding to the leafs of the tree rooted at $ancestor_{T_1}(i,\ell)$.*

A similar property holds for the second tree.

Lemma 3.10 *If a process reads at state s $ancestor_{T_2}(i,\ell) = v$, then in s, at least v processes have read the value n from $root_{T_1}$.*

Proof: By the code, a process reads the value 1 in the ith leaf of the second tree only if a process claiming ID i has read n at the root of the first tree. Since processes read only initialized values in the second tree, the lemma follows by induction on the levels. ∎

The crucial property which ensures the correctness of PIB is the following.

Lemma 3.11 *There are no pseudo-initialization steps after the first time that a process reads $root_{T_2} = n$.*

Proof: By Lemma 3.9, if any process reads $root_{T_2} = n$, then all processes have read $root_{T_1} = n$. By the code, when a process reads $root_{T_1} = n$, it does not take any pseudo-initialization steps in the first tree. And since no process marks its leaf in the second tree as 1 before it completes its pseudo-initialization steps in the second tree, the lemma follows. ∎

We can conclude with the following theorem.

Theorem 3.12 *There exists an algorithm for the Processor Identity Problem for n processes in the dirty memory model with expected running time $O(\log n)$, which requires $O(n)$ shared bits.*

Proof: As before, the algorithm consists of inserting a complete iteration of the algorithm of Figure 1 between any two memory accesses of the algorithm in Figure 4. It is not difficult to verify that Lemma 3.5 holds for the first and second tree independently even with the initialization steps, using the same arguments. We can conclude that $O(\log n)$ time units after the algorithm starts, the set of claimed IDs will stabilize; by Lemma 3.11 after additional $O(\log n)$ time units all processes will read a value of n in the root of the first tree; additional $O(\log n)$ time units are required until all processes finish initializing their respective leaf-root paths in the seconds tree; and after $O(\log n)$ additional time units, all processes wil halt with unique IDs. ∎

4 Necessary Conditions for Solving PIP

In this section we show that in the read-write model, no terminating algorithms exist for PIP if either n is not known in advance, or if the schedule is adaptive. These results hold even in the case where the memory is initialized. We also argue that there is no protocol for PIP that terminates in $o(\log n)$ time. All these impossibility results are based on the observation that at least one of the processes needs to "communicate" (not necessarily directly) with all the other processes before termination. We remark that this observation translates fairly easily into rigorous proofs of the time lower bound and the necessity of knowledge of n; the proof of impossibility for adaptive adversaries is more involved.

We analyze systems for PIP in terms of Markov chains [5]. Consider a single process taking steps according to a given PIP algorithm. (We consider only one process taking steps without interference of other processes.) That process can be viewed as a Markov chain, whose state is characterized by its local state and the state of the shared memory. We represent that Markov chain as a directed graph, whose nodes are the states of the Markov chain, and a directed edge connects two nodes iff the probability of transition from one to the other is positive. Given an algorithm $\mathcal{A}$ for PIP, this Markov chain is completely determined. Note that the symmetry condition implies that all processes have identical Markov graphs. Henceforth, we denote the graph corresponding to an algorithm $\mathcal{A}$ by $G_{\mathcal{A}}$, and we use the terms "states" and "nodes" interchangeably. One of the nodes (called *source node*), corresponds to the initial state of the system. A node is called *reachable* if there is a directed path to it from the source node. We say that a global state s^* *extends* a node s with respect to a process p_i if the local state of p_i in s and the state of the shared memory in s^* are as in s. We start with a general lemma for PIP in the read-write model.

Lemma 4.1 *Let s be any reachable node in $G_{\mathcal{A}}$, and let s^* be the global state of the n-process system with the same shared state portion as in s, and such that the local states of all processes are identical to the local state portion in s. Then there exists an oblivious schedule under which the system reaches s^* with positive probability.*

Proof: We show that the "round robin" schedule satisfies the requirement. Let s_0 be the source node in $G_{\mathcal{A}}$. We prove the lemma by induction on the distance d of s from s_0 in $G_{\mathcal{A}}$. The base case, $d = 0$, follows (with probability 1) from the symmetry requirement for PIP: s^* is the state where the shared memory is in the same state as in s, and all the processes are in the same state as in s. For the inductive step, suppose that s is at distance $d+1$ from s_0. Let s' be the node in $G_{\mathcal{A}}$ which is reachable in d steps from s_0, and such that s is reachable in one step from s'. By the inductive hypothesis, the global state corresponding to the symmetric combination of the s' nodes is reachable with some probability $\alpha > 0$. By the definition of $G_{\mathcal{A}}$, s is reachable in a single step of process p_i, with some probability $\beta > 0$, for all $1 \leq i \leq n$. Now consider scheduling exactly one step of each process. Since the processes take their steps when they are in identical local states, it must be the case that they either all read or all write in their respective additional step. Hence, their steps cannot influence one another, which means that the probability distributions of their next state are *independent*. Therefore, with probability $\alpha\beta^n > 0$, the global state s^*, is reached, and the inductive step is complete. ∎

Notice that Lemma 4.1 holds even for infinite state algorithms (so long as no zero probability transitions are ever taken).

Using Lemma 4.1, we can now prove the first necessary condition for Las-Vegas PIP algorithms.

Theorem 4.2 *There is no algorithm for PIP that terminates with probability* 1 *and works with unknown number of processes, even for oblivious schedules.*

Proof: By contradiction. Suppose, for simplicity, that $\mathcal{A}$ works for $n = 1$ and $n = 2$. (The argument extends directly to arbitrary different values of n.) We argue that in this case, $\mathcal{A}$ cannot terminate when run with a single process. For suppose that $\mathcal{A}$ terminates: let ρ be any terminating execution in which only one process takes steps. By Lemma 4.1, there exists an execution ρ' of positive probability with two processes such that both processes reach the same state as in the end of ρ. But since the last state in ρ is a terminating state, we conclude that both processes terminate in ρ', and since they are in identical local state, they must have the same output value, a contradiction to the uniqueness requirement. ∎

Again, we remark that Theorem 4.2 holds also for infinite memory algorithms.

We now turn to consider the case of adaptive adversaries. Intuitively, an adaptive adversary picks the processes to take steps based on the history of the system, or, more precisely, on the history of the shared portion of the system. (That is, the result holds even if the adversary has no access to the local states of the processes.) The following theorem implies that there is no finite-state Las-Vegas algorithm for adaptive adversaries.

Theorem 4.3 *There is no finite-state algorithm for the Processor Identity Problem that terminates with probability* 1 *if the schedule is adaptive, even if* n *is known.*

Proof: By contradiction. Suppose that a given algorithm $\mathcal{A}$ terminates with probability 1. Then for all $\epsilon > 0$ there exists T_ϵ such that the probability of $\mathcal{A}$ terminating in T_ϵ or less time units is larger than $1-\epsilon$. We derive a contradiction by showing that there exists $\epsilon_0 > 0$ (that depends only on $\mathcal{A}$ and n), such that for any given time T, there exists an adaptive schedule in which $\mathcal{A}$ cannot terminate in T time units with probability greater than $1 - \epsilon_0$.

The idea, as in the proof of Theorem 4.2, is to keep the processes "hidden" from each other. For simplicity of presentation, let us consider the case of $n = 2$. The proof can be extended to an arbitrary number of processes in a straightforward way.

Our first step is to decompose the corresponding Markov chain into irreducible subchains. In graph theoretic language, consider the Markov graph $G_\mathcal{A}$: it is a directed graph; we decompose it into strongly-connected components. That is, we partition the nodes into equivalence classes ("strong components"), such that two nodes s and s' are in the same class if and only if there is a directed path in $G_\mathcal{A}$ from s to s' and from s' to s. Given this decomposition, we define a *terminal component* to be a strong component such that no other component is reachable from it. Notice that the existence of terminal components in the Markov graphs is guaranteed by the fact that the number of nodes in $G_\mathcal{A}$ (i.e., the number of states of the algorithm $\mathcal{A}$) is finite. We now use a simple fact from the theory of Markov chains.

Lemma 4.4 *Let* s *be a state in a terminal component of* $G_\mathcal{A}$*, and let* s^* *be any global state that extends* s *with respect to some process* p_i*. Then for all* $\gamma < 1$ *there exists a positive integer* M_γ*, such that in an execution that starts at* s^* *and consists of* M_γ *steps of* p_i *alone,* s^* *occurs again with probability at least* γ*.*

Proof of Lemma: The state of a process p_i in an execution that starts from a state in a terminal component, and in which only p_i takes steps, can be represented by an irreducible Markov chain. The lemma follows from the fact that for a finite irreducible Markov chain, the expected recurrence time of any state is finite. ∎

Lemma 4.4 gives rise to the following immediate corollary.

Corollary 4.5 *Let* s *be a state in a terminal component of* $G_\mathcal{A}$*, let* s^* *be any global state that extends* s *with respect to some process* p_i*, and let* $\bar{v}$ *be the state of the shared portion of* s^**. Then for all* $\gamma < 1$ *there exists a positive integer* M_γ*, such that in an execution that starts at* s^* *and consists of* M_γ *steps of* p_i *alone,* $\bar{v}$ *occurs again with probability at least* γ*.*

We now continue with the proof of Theorem 4.3, by describing a complete strategy of an adaptive adversary, given an algorithm $\mathcal{A}$ and a time bound T. First, an arbitrary reachable state s in a terminal component of $G_\mathcal{A}$ is chosen. (This can be done since the algorithm completely characterizes $G_\mathcal{A}$.) The adversary then uses the round-robin schedule outlined in the proof of Lemma 4.1, which guarantees, with some probability $\alpha > 0$, that the corresponding symmetric global state s^*, in which all the processes are in the same local state as in s, and the shared memory is in the same state as the shared portion of s. We show that for any given time T, there is an adaptive schedule such that the algorithm fails to terminate in T time units with probability greater than $\alpha/5$.

We do this as follows. Let $\gamma = 1 - 1/2T$. Denote the state of the shared portion in s^* by $\bar{v}$. The adversary now lets p_1 take steps (at least one) until the values of the shared variables are again as specified by $\bar{v}$, or until M_γ steps (as obtained from Corollary 4.5) have been taken. This can be done since by our assumption that the adversary is adaptive, the adversary can "see" when $\bar{v}$ recurs. Moreover, Corollary 4.5 guarantees that this process will succeed with probability at least γ. When $\bar{v}$ recurs, the adversary lets p_2 take steps (again, at least one and no more than M_γ), until the configuration of the shared memory is $\bar{v}$ again. This can be done by the same reasoning as for p_1. Notice that now, p_1 and p_2 are *not* necessarily in the same state; however, p_2 is completely "hidden" from p_1, since the shared memory is in exactly the same state in which p_1 stopped taking steps. Therefore, the adversary can resume p_1 now, and p_1 must act as if it is the only process in the system.

Observe that this procedure of letting one process take steps until $\bar{v}$ is reached and then switch to the other process, can be repeated $2T$ times (thus resulting in a schedule with running time T), with probability of success at least

$$\gamma^{2T} = \left(1 - \frac{1}{2T}\right)^{2T} > \frac{1}{5} \qquad \text{for } T \geq 1 \ .$$

We can now complete the proof of Theorem 4.3. We argue that for any given $T > 0$, using the schedule specified above we get, with probability at least $\epsilon_0 = \alpha/5$, an execution in which neither p_1 nor p_2 can terminate in T time units. This is true because with probability at least α, s^* is reached, and with probability greater than $1/5$, once s^* is reached, p_1 and p_2 remain "hidden" from each other for T time units. Now we claim that termination of any of the processes in this execution implies a contradiction: since p_1 (say) did not observe any action of p_2, it follows that from the point of view of p_1, there exists an indistinguishable execution ρ in which p_2 is advancing in "lockstep" with p_1, maintaining symmetrical local state. If p_1 terminates in the given execution, then in ρ p_1 and p_2 terminate also, violating the uniqueness requirement. Since the uniqueness property must be met *always*, we have reached a contradiction. ∎

To show the necessity of the bounded space condition in Theorem 4.3, we have the following theorem.

Theorem 4.6 *There exists an unbounded-space algorithm for PIP that terminates with probability* 1 *under any fair adversary.*

The proof consists of an unbounded-state algorithm; the algorithm is a simple variant of the dynamic algorithm, where the contents of each register is simply a complete history of all the random signatures. We present a simplified version of it in Figure 5, where all shared registers are initialized by all processes, and termination is detected by scanning all the shared registers. The analysis is straightforward, and therefore omitted.

Our last result for this section is the simple observation that any algorithm for PIP requires $\Omega(\log n)$ time units.

Theorem 4.7 *There is no algorithm for PIP (including Monte-Carlo algorithms) that terminates in*

```
Shared Variables
  D          : a vector of N integers, initially all 0

Local Variables
  ID         : output value, initially random({1, ..., N})
  old_sign   : an integer, initially 0

Code
D[i] ← 0 for all 0 ≤ i < N
repeat
   either, with probability 1/2, do
      D[ID] ← old_sign ← random({0, 1}) + 2 · old_sign
   or, with probability 1/2, do
      if old_sign ≠ D[ID] then
         old_sign ← ⊥
         ID ← random({0, ..., N − 1})
         D[ID] ← old_sign ← random({0, 1}) + 2 · old_sign
until | {i : D[i] ≠ 0} | = n
ID ← | {i : D[i] ≠ 0 and i ≤ ID} |
```

Figure 5: Unbounded space algorithm for PIP under any adversary.

$o(\log n)$ *expected time.*

Proof: We show that for any algorithm there exists a schedule such that the expected running time under this schedule is at least $\log n$. Fix an execution. Define the set of *influencing processes* for a process p_i at state s_t, denoted $S_i(t)$, for all i and t, inductively as follows. At the initial state, we define $S_i(0) = \{p_i\}$ for all i. Suppose now that $S_i(t')$ is defined for all the processes and all steps $t' < t$, and consider the action a_t leading to state s_t. If a_t is a write action, then $S_i(t) = S_i(t-1)$ for all i. If a_t is a read action of a process p_k, say, let p_j the last process that wrote that register (if such p_j exists), and suppose this write occurred at action a_w. In this case we define $S_k(t) = S_k(t-1) \cup S_j(w)$, and $S_i(t) = S_i(t-1)$ for all $i \neq k$. If no such p_j exists, define $S_i(t) = S_i(t-1)$ for all i. Intuitively, the influencing set of a process at a state is the set of all processes that have communicated with that process directly or indirectly.

We claim that in any given execution ρ of a algorithm for PIP, a process may terminate only when its influencing set is $\{p_1, \ldots, p_n\}$. For suppose not, i.e., p_i terminates at step t with $p_j \notin S_i(t)$. Then there exists another positive-probability execution ρ', indistinguishable from ρ for p_i, in which p_j has the same random choices as p_i has, and in which p_i and p_j advance in lockstep. Clearly, p_i and p_j maintain identical local state in ρ', which in turn is identical to the state of p_i in ρ. Hence termination of p_i in ρ implies termination of p_i and p_j in ρ' with the same output value, a contradiction to the uniqueness requirement.

Now consider the executions resulting from the round robin schedule, where each step takes exactly one time unit. An easy induction on time shows that in these executions, $|S_i(t+1)| \leq 2|S_i(t)|$ for all processes p_i and steps t. Since we must have $|S_i(t)| = n$ at the terminating state for all processes, we conclude that the expected worst-case running time of every algorithm for PIP is $\Omega(\log n)$. ▮

5 PIP and the Read-Modify-Write Model

<u>Shared Variable</u>

$\quad$ *message* : takes values from $\{\texttt{init}, \texttt{ready}, \texttt{accept}, 0, 1, \texttt{ack}, \texttt{end}\}$, initially $\texttt{init}$

<u>Local Variables</u>

$\quad$ *ID* : output value

$\quad$ *mode* : takes values from $\{\mathit{start}, \mathit{get}, \mathit{seek}, \mathit{give}, \mathit{done}\}$, initially *start*

$\quad$ *rem*, k : integers, initially 0

<u>Code</u>

```
repeat
  case mode
    start: if message = init then                                   first access for anyone
               mode ← seek, ID ← 0, message ← ready
           else if message = ready then                             first access for others
               mode ← get, ID ← 0, message ← accept
    seek:  if message = accept then                                 new process detected
               mode ← give, message ← ID mod 2, rem ← ⌊ID/2⌋
    get:   if message ∈ {0, 1} then                                 receive last ID bit by bit
               ID ← ID + message · 2^k, k ← k + 1, message ← ack
           else if message = end then                               got the whole last ID
               ID ← ID + 1, mode ← seek, message ← ready
    give:  if message = ack then                                    last bit received, transmit next one
               if rem ≠ 0 then
                   message ← rem mod 2, rem ← ⌊rem/2⌋
               else mode ← done
                   message ← end
  end case
until mode = done
```

Figure 6: Deterministic algorithm for PIP in the read-modify-write model, using 3 shared bits

In this section we give positive and negative results for PIP in the read-modify-write model. First, we show that in this model, PIP can be solved deterministically using constant size memory, under any fair schedule. This result should be contrasted with the impossibility results of Theorems 4.2 and 4.3 for the read-write model. Our second result for this section shows that if the initial state of the protocol is arbitrary (i.e., if the self-stabilization model is assumed), then there is no protocol (including randomized protocols), that solves PIP with probability 1. This result uses the technique of Theorem 4.3.

Let us start with an informal description of a protocol for PIP that uses $\log N$ bits. We will then derive our constant-space protocol. The protocol using $\log N$ bits is trivial: the shared variable is used as a counter, or a"ticket dispenser", in the following sense. When a process enters the system, it accesses the variable, takes its current value to be its ID, and in the same step, increments the value of the variable by 1. It is straightforward to verify that this indeed produces unique IDs at the processes under any fair schedule.

In our constant space protocol, we still employ this "serial counter" approach. However, to reduce space, we use the shared variable as a "pipeline" to transmit counter values from one process to another, while the counter value is maintained in the *local memory* of the processes. Intuitively, there is some process "in charge" at any given time, which knows the current value of the counter. Whenever a new process p enters the system, it writes a request message in the shared variable. The process in charge responds by transmitting the current contents of the counter, bit by bit, with an acknowledgment for each bit. By the end of this procedure, p has the value of the counter; p then increments it by 1, takes it to be its ID, and becomes the process in charge. This serial style dialog will not be interrupted by other processes, due to the read-modify-write assumption. We assume that the shared variable is initialized with a special value, that tells whoever accesses the variable first, that it is in charge, and that the counter value is 0. We summarize with the following theorem.

Theorem 5.1 *In the read-modify-write model, there exists a deterministic protocol for PIP that requires a shared variable of 3 bits and works under any fair schedule.*

5.1 Impossibility for Self-Stabilizing Protocols

An algorithm is called *self-stabilizing* if it works correctly regardless of its initial state. It is straightforward to see that no self-stabilizing algorithm can ever halt: otherwise, a process may be put in a terminating state with erroneous output value as its first state. However, it is not immediately clear that a dynamic algorithm is not possible. Indeed, the dynamic algorithm of Figure 1 is self-stabilizing—under the assumption that the schedule is oblivious. The next result shows that the obliviousness of the schedule is required even when we have read-modify-write registers, if the algorithm has a finite state space.

Theorem 5.2 *There is no finite state, self-stabilizing algorithm that solves PIP in the read-modify-write registers with probability* 1 *if the schedule is adaptive.*

The proof is nearly identical to the proof of Theorem 4.3, and we therefore omit it. We remark only that the self-stabilization assumption serves as a substitute for Lemma 4.1, which does not hold in the read-modify-write model.

Acknowledgment

We thank Oded Goldreich for many insightful comments on an earlier draft of the paper. The third author would like to thank also Nancy Lynch and Yishay Mansour for many helpful discussions.

References

[1] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Daphne Koller, David Peleg, and Rüdiger Reischuk. Achievable cases in an asynchronous environment. In *28th Annual Symposium on Foundations of Computer Science, White Plains, New York*, pages 337–346, 1987.

[2] James E. Burns. Symmetry in systems of asynchronous processes. In *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York*, pages 169–174, 1981.

[3] Benny Chor, Amos Israeli, and Ming Li. On processor coordination using asynchronous hardware. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

[4] Ömer Eğecioğlu and Ambuj K. Singh. Naming symmetric processes using shared variables. Unpublished manuscript, 1992.

[5] W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. Wiley, 3rd edition, 1968.

[6] Ralph E. Johnson and Fred B. Schneider. Symmetry and similarity in distributed systems. In *Proceedings of the 4th ACM Symp. on Principles of Distributed Computing*, pages 13–22, 1985.

[7] Leslie Lamport. On interprocess communication (part II). *Distributed Computing*, 1(2):86–101, 1986.

[8] Tom Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes.* Morgan-Kaufman, 1991.

[9] Richard J. Lipton and Arvin Park. The processor identity problem. *Info. Proc. Lett.*, 36:91–94, October 1990.

[10] Yishay Mansour. Personal communication, 1992.

[11] Shang-Hua Teng. The processor identity problem. *Info. Proc. Lett.*, 34:147–154, April 1990.