Anonymous Processors with Synchronous Shared Memory: Monte Carlo Algorithms*

Bogdan S. Chlebus[1], Gianluca De Marco[2], and Muhammed Talo[3]

1 **Department of Computer Science and Engineering, University of Colorado Denver, Denver, Colorado 80217, USA**
bogdan.chlebus@ucdenver.edu

2 **Dipartimento di Informatica, Università degli Studi di Salerno, Fisciano, 84084 Salerno, Italy**
demarco@dia.unisa.it

3 **Bilgisayar Mühendisliği, Munzur Üniversitesi, 62000 Tunceli, Turkey**
muhammedtalo@munzur.edu.tr

Abstract

We consider synchronous distributed systems in which processors communicate by shared read-write variables. Processors are anonymous and do not know their number n. The goal is to assign individual names by all the processors to themselves. We develop algorithms that accomplish this for each of the four cases determined by the following independent properties of the model: concurrently attempting to write distinct values into the same shared memory register either is allowed or not, and the number of shared variables either is a constant or it is unbounded. For each such a case, we give a Monte Carlo algorithm that runs in the optimum expected time and uses the expected number of $\mathcal{O}(n \log n)$ random bits. All our algorithms produce correct output upon termination with probabilities that are $1 - n^{-\Omega(1)}$, which is best possible when terminating almost surely and using $\mathcal{O}(n \log n)$ random bits.

1998 ACM Subject Classification C.1.4 Parallel Architectures, F.1.1 Models of Computation, F.1.2 Modes of Computation

Keywords and phrases anonymous processors, synchrony, shared memory, read-write registers, naming, Monte Carlo algorithms

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.15

1 Introduction

We study naming algorithms in distributed systems consisting of anonymous processors that communicate by reading from and writing to shared memory. We say that a parameter of an algorithmic problem is *known* when it can be used in a code of algorithm. We restrict our attention to synchronous systems and do not assume that the number of processors n is known.

The model of synchronous systems with read-write registers is known as the Parallel Random Access Machine (PRAM). It is a generalization of the Random Access Machine model of sequential computation [16] to the realm of synchronous concurrent processing.

We consider two categories of naming problems depending on how much shared memory is available for a PRAM. In one case, a constant number of memory cells is available. This means that the amount of memory is independent from the number of processors n but as

* The full version of this paper merged with [12] is available as [13], https://arxiv.org/abs/1507.02272.

Table 1 Four naming problems, as determined by the PRAM model and the available amount of shared memory, with the respective performance bounds of their solutions as functions of the number of processors n. When time is marked as "polylog" this means that the algorithm comes in two variants, such that in one the expected time is $\mathcal{O}(\log n)$ and the amount of used shared memory is suboptimal $n^{\mathcal{O}(1)}$, and in the other the expected time is suboptimal $\mathcal{O}(\log^2 n)$ but the amount of used shared memory misses optimality by at most a logarithmic factor.

PRAM Model	Memory	Time	Algorithm
Arbitrary	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Arbitrary-Bounded-MC in Section 3
Arbitrary	unbounded	polylog	Arbitrary-Unbounded-MC in Section 4
Common	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$	Common-Bounded-MC in Section 5
Common	unbounded	polylog	Common-Unbounded-MC in Section 6

large as needed in an algorithm's design. In the other case, the amount of shared memory cells is unlimited, and how much is used by an algorithm depends on n. When an unbounded amount of memory cells is assumed to be available, then the expected number of memory cells that are actually used is considered as a performance metric.

Independently of the amount of shared memory available, we consider the two versions of the naming problems that are determined by the semantics of concurrent writing. This is represented by the corresponding PRAM variants, which are either the Arbitrary PRAM or the Common PRAM.

Naming in the PRAM model is also considered in a companion paper [12], which is about the same problem to assign names for the anonymous processors, with the only difference that n is assumed to be known in [12], while we assume in this paper that n is unknown. Whether n is known or not is reflected in the classes of algorithms we develop in these two papers: these are Monte Carlo algorithms for an unknown n in this paper, and Las Vegas algorithms for a known n in [12].

A summary of the results

We consider four naming problems in synchronous read-write shared memory. They are determined by two independent specifications the naming problems: the amount of shared memory and the PRAM's variant.

The naming algorithms we give terminate with probability 1 and are all Monte Carlo. Each algorithm uses the optimum expected number $\mathcal{O}(n \log n)$ of random bits. We show that a Monte Carlo naming algorithm that uses $\mathcal{O}(n \log n)$ random bits has to have the property that it fails to assign unique names with the probability that is $n^{-\Omega(1)}$. All Monte Carlo algorithms that we give have the optimum polynomial probability of error. The list of the naming problems' specifications and the corresponding algorithms with their performance bounds are summarized in Table 1. Most proofs are omitted; they can be found in [13].

Previous and related work

The naming problem for a synchronous PRAM has not been previously considered in the literature, to the best of the authors' knowledge, except for the companion paper [12]. The problem of concurrent communication in anonymous networks was first considered by

Angluin [3]. That work showed, in particular, that randomization is needed in naming algorithms when executed in environments that are perfectly symmetric; other related impossibility results are surveyed by Fich and Ruppert [18]. There is a voluminous literature on various aspects of computing and communication in anonymous systems, we concentrate on the related topics for anonymous *asynchronous* distributed shared-memory systems.

We begin with work on naming in shared-memory systems with read-write registers. Lipton and Park [23] considered naming in asynchronous distributed systems with read-write shared memory controlled by adaptive schedulers; they proposed a solution that terminates with positive probability, and which can be made arbitrarily close to 1 assuming that n is known. Eğecioğlu and Singh [15] proposed a polynomial-time Las Vegas naming algorithm for asynchronous systems with known n and read-write shared memory with oblivious scheduling of events. Kutten et al. [22] provided a thorough study of naming in asynchronous systems of shared read-write memory. They gave a Las Vegas algorithm for an oblivious scheduler for the case of known n, which works in the expected time $\mathcal{O}(\log n)$ while using $\mathcal{O}(n)$ shared registers, and also showed that a logarithmic time is required to assign names to anonymous processes. Additionally, they showed that if n is unknown then a Las Vegas naming algorithm does not exist, and a finite-state Las Vegas naming algorithm can work only for an oblivious scheduler. Panconesi et al. [24] gave a randomized wait-free naming algorithm in anonymous systems with processes prone to crashes that communicate by single-writer registers. The model considered in that work assigns unique single-writer registers to nameless processes and so has a potential to defy the impossibility of wait-free naming for general multi-writer registers proved by Kutten et al. [22]. Buhrman et al. [11] considered the relative complexity of naming and consensus problems in asynchronous systems with shared memory that are prone to crash failures, demonstrating that naming is harder that consensus.

Now we review work on problems in anonymous distributed systems different from naming. Aspnes et al. [4] gave a comparative study of anonymous distributed systems with different communication mechanisms, including broadcast and shared-memory objects of various functionalities, like read-write registers and counters. Alistarh et al. [2] gave randomized renaming algorithms that act like naming ones, in that process identifiers are not referred to; for more or renaming see [1, 6, 14]. Aspnes et al. [5] considered solving consensus in anonymous systems with infinitely many processes. Attiya et al. [7] and Jayanti and Toueg [21] studied the impact of initialization of shared registers on solvability of tasks like consensus and wakeup in fault-free anonymous systems. Bonnet et al. [10] considered solvability of consensus in anonymous systems with processes prone to crashes but augmented with failure detectors. Guerraoui and Ruppert [19] showed that certain tasks like time-stamping, snapshots and consensus have deterministic solutions in anonymous systems with shared read-write registers prone to process crashes. Ruppert [25] studied the impact of anonymity of processes on wait-free computing and mutual implementability of types of shared objects.

A systematic exposition of shared-memory algorithm can be found in [8], when approached from the distributed-computing perspective, and in [20], when approached from the parallel-computing one. General questions of computability in anonymous message-passing systems implemented in networks were studied by Boldi and Vigna [9], Emek et al. [17], and Sakamoto [26].

2 Technical Preliminaries

Two operations are said to be performed concurrently when they are invoked in the same round of an execution of a PRAM. We assume that concurrent reading from a memory cell and writing to this same memory cell never occur. This can be made without loss of generality for a synchronous PRAM because we can partition an execution into alternating "writing" and "reading" rounds, which results in slowing the execution by at most a factor of 2. The meaning of concurrent reading from the same memory cell is straightforward, in that all the readers get the value stored in this memory cell.

Concurrent writing to the same memory location needs to be further clarified.

When multiple writers want to write the same value each in the same round to the same memory cell, then we should assume that this value gets written. This scenario is so enticing, that it leads to a PRAM variant called *Common*, for which it is assumed that only such concurrent writes are legitimate, in that an attempt to write different values concurrently to the same memory location results in a runtime error.

On the other hand, not having to worry about consistency of written values is also attractive, which leads to a PRAM variant called *Arbitrary*. This model allows any admissible value to be attempted to be written concurrently. A downside is that the model does not determine the outcome of a write but only that one of the values gets written. A consequence is that when we argue about correctness then all possible selections among the attempted values as actually written successfully need to be considered.

Balls into bins

In the course of probabilistic analysis of algorithms, we will often model actions of processors by throwing balls into bins. This can be done in two natural ways. One is such that memory addresses are interpreted as bins and the values written represent balls, possibly with labels. Then total number of balls considered will always be n, that is, be equal to the number of processors of a PRAM. Another possibility is when bins represent rounds and selecting a bin results in performing a write to a suitable shared register in the respective round.

Throwing balls into bins will be performed repeatedly in each instance of modeling the behavior of an algorithm. Each instance of throwing a number of balls into bins is then called a *stage*. There will be an additional numeric parameter $\beta > 0$, and we call the process of throwing balls into bins the *β-process*, accordingly. This parameter β may determine the number of bins in a stage and also when a stage is the last one in an execution of the β-process.

When we sum up the numbers of available bins over all the stages of an execution of a β-process until termination, then the result is *the number of bins ever needed* in this execution. Similarly, *the number of bits ever generated* in an execution of a β-process is the sum of all the numbers of random bits needed to be generated to place balls, over all the stages and balls until termination of this execution.

Verifying collisions

We will use a randomized procedure for Common PRAM to verify if a collision occurs in a bin. This procedure Verify-Collision was given in [12]; it is represented in Figure 1 for a direct reference. Bins are interpreted in two different ways. When algorithms use a constant number of shared memory registers, then bins are typically interpreted as future rounds during which verification for a collision will be performed, one verification in $\mathcal{O}(1)$

Procedure VERIFY-COLLISION (x)

```
initialize Heads[x] ← Tails[x] ← false
toss_v ← outcome of tossing a fair coin
if toss_v = tails
   then Tails[x] ← true
   else Heads[x] ← true
return Tails[x] = Heads[x]
```

Figure 1 A pseudocode for a processor v of a Common PRAM, where x is a positive integer. `Heads` and `Tails` are arrays of shared memory cells. When the parameter x is dropped in a call then this means that $x = 1$. The procedure returns `true` when a collision has been detected.

rounds. In such a scenario, procedure VERIFY-COLLISION is used without a parameter, because just one shared memory register is needed to carry out one verification. When algorithms use an unbounded array of shared registers, then bins are typically interpreted as some designated shared registers. In such a scenario, procedure VERIFY-COLLISION is invoked with a parameter indicating which bin is verified for collision, because multiple verifications for collisions in different bins can be performed concurrently.

▶ **Lemma 1** ([12, 13]). *For an integer x, procedure* VERIFY-COLLISION (x) *executed by one processor never detects a collision, and when multiple processors execute this procedure then a collision is detected with probability at least $\frac{1}{2}$.*

Properties of naming algorithms

Randomized naming algorithms are categorized as either Monte Carlo or Las Vegas, which are defined as follows. A randomized algorithm is *Las Vegas* when it terminates almost surely and the algorithm returns a correct output upon termination. A randomized algorithm is *Monte Carlo* when it terminates almost surely and an incorrect output may be produced upon termination, but the probability of error converges to zero with the size of input growing unbounded. The naming algorithms we develop are all Monte Carlo and have the probability of error converging to zero with a rate that is polynomial in n. Moreover, when incorrect names are assigned, then the set of integers used as names makes a contiguous segment starting from the smallest name 1 and the only possible kind of error is that duplicate names are given.

A naming algorithm cannot be Las Vegas when n is unknown, as was observed by Kutten et al. [22] for asynchronous computations against an oblivious adversary. An analogous fact holds for synchronous computations.

▶ **Proposition 1.** *There is no Las Vegas naming algorithm for a PRAM with $n > 1$ processors that does not refer to the number of processors n in its code.*

Proof. Let us suppose, to arrive at a contradiction, that such a naming Las Vegas algorithm exists. Consider a system of $n-1 \geq 1$ processors, and an execution $\mathcal{E}$ on these $n-1$ processors that uses specific strings of random bits such that the algorithm terminates in $\mathcal{E}$ with these random bits. Such strings of random bits exist because the algorithm terminates almost surely.

Let v_1 be a processor that halts latest in $\mathcal{E}$ among the $n-1$ processors. Let $\alpha_{\mathcal{E}}$ be the string of random bits generated by processor v_1 by the time it halts in $\mathcal{E}$. Consider an execution $\mathcal{E}'$ on $n \geq 2$ processors such that n processors obtain the same strings of random bits as in $\mathcal{E}$ and an extra processor v_2 obtains $\alpha_{\mathcal{E}}$ as its random bits. The executions $\mathcal{E}$ and $\mathcal{E}'$ are indistinguishable for the $n-1$ processors participating in $\mathcal{E}$, so they assign themselves the same names and halt. Processor v_2 performs the same reads and writes as processor v_1 and assigns itself the same name as processor v_1 does and halts in the same round as processor v_1. This is the termination round because by that time all the other processor have halted as well.

It follows that execution $\mathcal{E}'$ results in a name being duplicated. The probability of duplication for n processors is at least as large as the probability to generate two identical finite random strings in $\mathcal{E}'$ for some two processors, so this probability is positive. ◀

We give algorithms that use the expected number of $\mathcal{O}(n \log n)$ random bits with large probability. The following fact allows to argue about their optimality with respect to the number of random bits.

▶ **Proposition 2** ([12, 13]). *If a randomized naming PRAM algorithm executed by n anonymous processors is correct with some probability p_n then it requires $\Omega(n \log n)$ random bits with the same probability p_n.*

If n is unknown, then the restriction $\mathcal{O}(n \log n)$ on the number of random bits makes it inevitable that the probability of error is at least polynomially bounded from below, as we show next.

▶ **Proposition 3.** *For unknown n, if a randomized naming algorithm is executed by n anonymous processors, then an execution is incorrect, in that duplicate names are assigned to distinct processors, with probability that is at least $n^{-\Omega(1)}$, assuming that the algorithm uses $\mathcal{O}(n \log n)$ random bits with probability $1 - n^{-\Omega(1)}$.*

Proof. Suppose the algorithm uses at most $cn \lg n$ random bits with probability p_n when executed by a system of n processors, for some constant $c > 0$. Then one of these processors uses at most $c \lg n$ bits with probability p_n, by the pigeonhole principle.

Consider an execution for $n+1$ processors. Let us distinguish a processor v. Consider the actions of the remaining n processors: one of them, say w, uses at most $c \lg n$ bits with the probability p_n. Processor v generates the same string of bits with probability $2^{-c \lg n} = n^{-c}$. The random bits generated by w and v are independent. Therefore duplicate names occur with probability at least $n^{-c} \cdot p_n$. When we have a bound $p_n = 1 - n^{-\Omega(1)}$, then the probability of duplicate names is at least $n^{-c}(1 - n^{-\Omega(1)}) = n^{-\Omega(1)}$. ◀

In gauging the optimality of performance of naming algorithms, we will refer to lower bounds on time of such algorithms that can be found in [12, 13], we restate them here for easy reference.

▶ **Theorem 2** ([12, 13]). *A randomized naming algorithm for a Common PRAM with n processors and $C > 0$ shared memory cells operates in $\Omega(n \log n/C)$ expected time when it is either a Las Vegas algorithm or a Monte Carlo algorithm with the probability of error smaller than $1/2$.*

▶ **Theorem 3** ([12, 13]). *A randomized naming algorithm for an Arbitrary PRAM with n processors and $C > 0$ shared memory cells operates in $\Omega(n/C)$ expected time when it is either a Las Vegas algorithm or a Monte Carlo algorithm with the probability of error smaller than $1/2$.*

The following fact holds for both Common and Arbitrary PRAMs.

▶ **Theorem 4** ([12, 13]). *A randomized naming algorithm for a PRAM with n processors operates in $\Omega(\log n)$ expected time when it is either a Las Vegas algorithm or a Monte Carlo algorithm with the probability of error smaller than $1/2$.*

3 Arbitrary with Bounded Memory

We develop a naming algorithm for an Arbitrary PRAM with a constant number of shared memory cells. The algorithm is called Arbitrary-Bounded-MC and its pseudocode is given in Figure 2.

The underlying idea is to have all processors repeatedly attempt to obtain tentative names and terminate when the probability of duplicate names is gauged to be sufficiently small. To this end, each processor writes an integer selected from a suitable "selection range" into a shared memory register and next reads this register to verify whether the write was successful or not. A successful write results in each such a processor getting a tentative name by reading and incrementing another shared register operating as a counter. One of the challenges here is to determine a selection range from which random integers are chosen for writing. A good selection range is large enough with respect to the number of writers, which is unknown, because when the range is too small then multiple processors may select the same integer and so all of them get the same tentative name after this integer gets written successfully. The algorithm keeps the size of a selection range growing with each failed attempt to assign tentative names.

There is an inherent tradeoff here, since on the one hand, we want to keep the size of used shared memory small, as a measure of efficiency of the algorithm, while, at the same time, the larger the range of memory the smaller the probability of collision of random selections from a selection range and so of the resulting duplicate names. Additionally, increasing the selection range repeatedly costs time for each such a repetition, while we also want to minimize the running time as the metric of performance. The algorithm keeps increasing the selection range with a quadratic rate, which turns out to be sufficient to optimize all the performance metrics we measure. The algorithm terminates when the number of selected integers from the current selection range makes a sufficiently small fraction of the size of the used range.

The structure of the pseudocode in Figure 2 is determined by the main repeat-loop. Each iteration of this loop begins with doubling the variable k, which determines the selection range $[1, 2^k]$. This means that the size of the selection range increases quadratically with consecutive iterations of the main repeat-loop. A processor begins an iteration of the main loop by choosing an integer uniformly at random from the current selection range $[1, 2^k]$. There is an inner repeat-loop, nested within the main loop, which assigns tentative names depending on the random selections just made.

All processors repeatedly write to a shared variable `Pad` and next read to verify if the write was successful. It is possible that different processors attempt to write the same value and then verify that their write was successful. The shared variable `Last-Name` is used to proceed through consecutive integers to provide tentative names to be assigned to the latest successful writers. When multiple processors attempt to write the same value to `Pad` and it gets written successfully, then all of them obtain the same tentative name. The variable `Last-Name`, at the end of each iteration of the inner repeat-loop, equals the number of occupied bins. The shared variable `All-Named` is used to verify if all processors have tentative names. The outer loop terminates when the number of assigned names, which is

Algorithm ARBITRARY-BOUNDED-MC

```
initialize k ← 1                                  /* initial approximation of lg n */
repeat
   initialize Last-Name ← name_v ← 0
   k ← 2k
   bin_v ← random integer in [1, 2^k]             /* throw a ball into a bin */
   repeat
      All-Named ← true
      if name_v = 0 then
         Pad ← bin_v
         if Pad = bin_v then
            Last-Name ← Last-Name + 1
            name_v ← Last-Name
         else
            All-Named ← false
   until All-Named
until Last-Name ≤ 2^(k/β)
```

Figure 2 A pseudocode for a processor v of an Arbitrary PRAM with a constant number of shared memory cells. The variables `Last-Name`, `All-Named` and `Pad` are shared. The private variable `name` stores the acquired name. The constant $\beta > 0$ is a parameter to be determined by analysis.

the same as the number of occupied bins, is smaller than or equal to $2^{k/\beta}$, where $\beta > 0$ is a parameter to be determined in analysis.

▶ **Theorem 5.** *Algorithm* ARBITRARY-BOUNDED-MC *always terminates, for any $\beta > 0$. For each $a > 0$ there exists $\beta > 0$ and $c > 0$ such that the algorithm assigns unique names, works in time at most cn, and uses at most $cn \ln n$ random bits, all this with probability at least $1 - n^{-a}$.*

Algorithm ARBITRARY-BOUNDED-MC is optimal with respect to the following performance measures: the expected time $\mathcal{O}(n)$, by Theorem 3, the expected number of random bits $\mathcal{O}(n \log n)$, by Proposition 2, and the probability of error $n^{-\mathcal{O}(1)}$, by Proposition 3.

4 Arbitrary with Unbounded Memory

We develop a naming algorithm for Arbitrary PRAM with an unbounded amount of shared registers. The algorithm is called ARBITRARY-UNBOUNDED-MC and its pseudocode is given in Figure 3.

The underlying idea is to parallelize the process of selection of names applied in Section 3 in algorithm ARBITRARY-BOUNDED-MC so that multiple processes could acquire information in the same round that later would allow them to obtain names. As algorithm ARBITRARY-BOUNDED-MC used shared registers `Pad` and `Last-Name`, the new algorithm uses arrays of shared registers playing similar roles. The values read-off from `Last-Name` cannot be used directly as names, because multiple processors can read the same values, so we need to distinguish between these values to assign names. To this end, we assign ranks

Algorithm ARBITRARY-UNBOUNDED-MC

```
initialize k ← 1                                   /* initial approximation of lg n */
repeat
   initialize All-Named ← true
   initialize position_v ← (0, 0)
   k ← r(k)
   bin_v ← random integer in [1, 2^k/(βk)]    /* choose a bin for the ball */
   label_v ← random integer in [1, 2^{βk}]    /* choose a label for the ball */
   for i ← 1 to βk do
      if position_v = (0, 0) then
         Pad [bin_v] ← label_v
         if Pad [bin_v] = label_v then
            Last-Name [bin_v] ← Last-Name [bin_v] + 1
            position_v ← (bin_v, Last-Name [bin_v])
   if position_v = (0, 0) then
      All-Named ← false
until All-Named
name_v ← the rank of position_v
```

Figure 3 A pseudocode for a processor v of an Arbitrary PRAM, when the number of shared memory cells is unbounded. The variables `Pad` and `Last-Name` are arrays of shared memory cells, the variable `All-Named` is shared as well. The private variable `name` stores the acquired name. The constant $\beta > 0$ and an increasing function $r(k)$ are parameters.

to processors based on their lexicographic ordering by pairs of numbers determined by `Pad` and `Last-Name`.

The pseudocode in Figure 3 is structured as a repeat-loop. In the first iteration, the parameter k equals 1, and in subsequent ones is determined by iterations of the increasing integer-valued function $r(k)$, which is a parameter. We consider two instantiations of the algorithm, determined by $r(k) = k + 1$ and by $r(k) = 2k$. In one iteration of the main repeat-loop, a processor uses two variables $\texttt{bin} \in [1, 2^k/(\beta k)]$ and $\texttt{label} \in [1, 2^{\beta k}]$, which are selected independently and uniformly at random from the respective ranges.

We interpret `bin` as a bin's number and `label` as a label for a ball. Processors write their values `label` into the respective bin by instruction $\texttt{Pad}\,[\texttt{bin}] \leftarrow \texttt{label}$ and verify what value got written. After a successful write, a processor increments $\texttt{Last-Name}[\texttt{bin}]$ and assigns the pair $(\texttt{bin}, \texttt{Last-Name}\,[\texttt{bin}])$ as its *position*. This is repeated βk times by way of iterating the inner for-loop. This loop has a specific upper bound βk on the number of iterations because we want to ascertain that there are at most βk balls in each bin. The main repeat-loop terminates when all values attempted to be written actually get written. Then processors assign themselves names according to the ranks of their positions. The array `Last-Name` is assumed to be initialized to 0's, and in each iteration of the repeat-loop we use a fresh region of shared memory to allocate this array.

▶ **Theorem 6.** *Algorithm* ARBITRARY-UNBOUNDED-MC *always terminates, for any* $\beta > 0$. *For each* $a > 0$, *there exists* $\beta > 0$ *and* $c > 0$ *such that the algorithm assigns unique names and has the following additional properties with probability* $1 - n^{-a}$. *If* $r(k) = k + 1$ *then*

at most $cn/\ln n$ *memory cells are ever needed,* $cn\ln^2 n$ *random bits are ever generated, and the algorithm terminates in time* $\mathcal{O}(\log^2 n)$*. If* $r(k) = 2k$ *then at most* $cn^2/\ln n$ *memory cells are ever needed,* $cn\ln n$ *random bits are ever generated, and the algorithm terminates in time* $\mathcal{O}(\log n)$*.*

The instantiations of algorithm Arbitrary-Unbounded-MC are close to optimality with respect to some of the performance metrics we consider, depending on whether $r(k) = k + 1$ or $r(k) = 2k$. If $r(k) = k + 1$ then the algorithm's use of shared memory would be optimal if its time were $\mathcal{O}(\log n)$, by Theorem 3, but as it is, the algorithm misses space optimality by at most a logarithmic factor, since the algorithm's running time is $\mathcal{O}(\log^2 n)$. Similarly, if $r(k) = k+1$ then the number of random bits ever generated $\mathcal{O}(n\log^2 n)$ misses optimality by at most a logarithmic factor, by Proposition 2. On the pother hand, if $r(k) = 2k$ then the expected time $\mathcal{O}(\log n)$ is optimal, by Theorem 4, the expected number of random bits $\mathcal{O}(n\log n)$ is optimal, by Proposition 2, and the probability of error $n^{-\mathcal{O}(1)}$ is optimal, by Proposition 3, but the amount of used shared memory misses optimality by at most a polynomial factor, by Theorem 3.

5 Common with Bounded Memory

Algorithm Common-Bounded-MC, which we present in this section, solves the naming problem for Common PRAM with a constant number of shared read-write registers. The algorithm has its pseudocode in Figure 6. To make the exposition of this algorithm more modular, we use two procedures Estimate-Size and Extend-Names. The pseudocodes of these procedures are given in Figures 4 and 5, respectively. The private variables in the pseudocode in Figure 6 have the following meaning: `size` is an approximation of the number of processors n, and `number-of-bins` determines the size of the range of bins we throw conceptual balls into.

The main task of procedure Estimate-Size is to produce an estimate of the number n of processors. Procedure Extend-Names is iterated multiple times, each iteration is intended to assign names to a group of processors. This is accomplished by the processors selecting integer values at random, interpreted as throwing balls into bins, and verifying for collisions. Each selection of a bin is followed by a collision detection. A ball placement without a detected collision results in a name assigned, otherwise the involved processors try again to throw balls into a range of bins. The effectiveness of the resulting algorithm hinges of calibrating the number of bins to the expected number of balls to be thrown.

Balls into bins for the first time

The role of procedure Estimate-Size, when called by algorithm Common-Bounded-MC, is to estimate the unknown number of processors n, which is returned as `size`, to assign a value to variable `number-of-bins`, and assign values to each private variable `bin`, which indicates the number of a selected bin in the range $[1, \texttt{number-of-bins}]$. The procedure tries consecutive values of k as approximations of $\lg n$. For a given k, an experiment is carried out to throw n balls into $k2^k$ bins. The execution stops when the number of occupied bins is at most 2^k, and then $3 \cdot 2^k$ is treated as an approximation of n and $k2^k$ is the returned number of bins.

▶ **Lemma 7.** *For* $n \geq 20$ *processors, procedure* Estimate-Size *returns an estimate* `size` *of* n *such that the inequality* $\texttt{size} < 6n$ *holds with certainty and the inequality* $n < \texttt{size}$ *holds with probability* $1 - 2^{-\Omega(n)}$*.*

Procedure ESTIMATE-SIZE

```
initialize k ← 2                                   /* initial approximation of lg n */
repeat
   k ← k + 1
   bin_v ← random integer in [1, k 2^k]
   initialize Nonempty-Bins ← 0
   for i ← 1 to k 2^k do
      if bin_v = i then
         Nonempty-Bins ← Nonempty-Bins + 1
until Nonempty-Bins ≤ 2^k
return (3 · 2^k, k 2^k)                  /* 3 · 2^k is size, k 2^k is number-of-bins */
```

Figure 4 A pseudocode for a processor v of a Common PRAM. This procedure is invoked by algorithm COMMON-BOUNDED-MC in Figure 6. The variable `Nonempty-Bins` is shared.

Procedure EXTEND-NAMES's behavior can also be interpreted as throwing balls into bins, where a processor v's ball is in a bin x when $\texttt{bin}_v = x$. The procedure first verifies the suitable range of bins $[1, \texttt{number-of-bins}]$ for collisions. A verification for collisions takes either just a constant time or $\Theta(\log n)$ time.

A constant verification occurs when there is no ball in the considered bin i, which is verified when the line "`if` $\texttt{bin}_x = i$ for some processor x" in the pseudocode in Figure 5 is to be executed. Such a verification is performed by using a shared register initialized to 0, into which all processors v with $\texttt{bin}_v = i$ write 1, then all the processors read this register, and if the outcome of reading is 1 then all write 0 again, which indicates that there is at least one ball in the bin, otherwise there is no ball.

A logarithmic-time verification of collision occurs when there is some ball in the corresponding bin. This triggers calling procedure VERIFY-COLLISION precisely $\beta \lg n$ times; notice that this procedure has the default parameter 1, as only one bin is verified at a time. Ultimately, when a collision is not detected for some processor v whose ball is the bin, then this processor increments `Last-Name` and assigns its new value as a tentative name. Otherwise, when a collision is detected, processor v places its ball in a new bin when the last line in Figure 5 is executed.

To prepare for the next round of throwing balls, the variable `number-of-bins` may be reset. During one iteration of the main repeat-loop of the pseudocode of algorithm COMMON-BOUNDED-MC in Figure 6, the number of bins is first set to a value that is $\Theta(n \log n)$ by procedure ESTIMATE-SIZE. Immediately after that, it is reset to $\Theta(n)$ by the first call of procedure EXTEND-NAMES, in which the instruction $\texttt{number-of-bins} \leftarrow \texttt{size}$ is performed. Here, we need to notice that $\texttt{number-of-bins} = \Theta(n \log n)$ and $\texttt{size} = \Theta(n)$, by the pseudocodes in Figures 4 and 6 and Lemma 7.

In the course of analysis of performance of procedure EXTEND-NAMES, we consider a balls-into-bins process; we call it simply the *ball process*. It proceeds through stages so that in a stage we have a number of balls which we throw into a number of bins. The sets of bins used in different stages are disjoint. The number of balls and bins used in a stage are as determined in the pseudocode in Figure 5, which means that there are n balls and the numbers of bins are as determined by an execution of procedure ESTIMATE-SIZE, that is,

Procedure EXTEND-NAMES

```
initialize Collision-Detected ← collision_v ← false
for i ← 1 to number-of-bins do
   if bin_x = i for some processor x then
     if bin_v = i then
       for j ← 1 to β lg size do
         if VERIFY-COLLISION then
           Collision-Detected ← collision_v ← true
       if not collision_v then
         Last-Name ← Last-Name + 1
         name_v ← Last-Name
         bin_v ← 0
if (number-of-bins > size) then
   number-of-bins ← size
if collision_v then
   bin_v ← random integer in [1, number-of-bins]
```

Figure 5 A pseudocode for a processor v of a Common PRAM. This procedure invokes procedure VERIFY-COLLISION, whose pseudocode is in Figure 1, and is itself invoked by algorithm COMMON-BOUNDED-MC in Figure 6. The variables `Last-Name` and `Collision-Detected` are shared. The private variable `name` stores the acquired name. The constant $\beta > 0$ is to be determined in analysis.

the first stage uses `number-of-bins` bins and subsequent stages use `size` bins, as returned by ESTIMATE-SIZE.

The only difference between the ball process and the actions of procedure EXTEND-NAMES is that collisions are detected with certainty in the ball process rather than being tested for. In particular, the parameter β is not involved in the ball process (nor in its name). The ball process terminates in the first stage in which no multiple bins are produced, so that there are no collisions among the balls.

▶ **Lemma 8.** *The ball process modeling the actions of procedure* EXTEND-NAMES *results in all balls ending single in their bins and the number of times a ball is thrown, summed over all the stages, being $\mathcal{O}(n)$, both events occurring with probability $1 - n^{-\Omega(\log n)}$.*

The following Theorem 9 summarizes the performance of algorithm COMMON-BOUNDED-MC (see the pseudocode in Figure 6) as a Monte Carlo one.

▶ **Theorem 9.** *Algorithm* COMMON-BOUNDED-MC *terminates almost surely. For each $a > 0$, there exists $\beta > 0$ and $c > 0$ such that the algorithm assigns unique names, works in time at most $cn \ln n$, and uses at most $cn \ln n$ random bits, each among these properties holding with probability at least $1 - n^{-a}$.*

Proof. One iteration of the main repeat-loop suffices to assign names with probability $1 - n^{-\Omega(\log n)}$, by Lemma 8. This means that the probability of not terminating by the ith iteration is at most $(n^{-\Omega(\log n)})^i$, which converges to 0 with i growing to infinity.

The algorithm returns duplicate names only when a collision occurs that is not detected by procedure VERIFY-COLLISION. For a given multiple bin, one iteration of this procedure

Algorithm COMMON-BOUNDED-MC

```
repeat
   initialize Last-Name ← 0
   (size, number-of-bins) ← ESTIMATE-SIZE
   for ℓ ← 1 to lg size do
      EXTEND-NAMES
      if not Collision-Detected then return
```

Figure 6 A pseudocode for a processor v of a Common PRAM, where there is a constant number of shared memory cells. Procedures ESTIMATE-SIZE and EXTEND-NAMES have their pseudocodes in Figures 4 and 5, respectively. The variables `Last-Name` and `Collision-Detected` are shared.

does not detect collision with probability at most $1/2$, by Lemma 1. Therefore $\beta \lg \texttt{size}$ iterations do not detect collision with probability $\mathcal{O}(n^{-\beta/2})$, by Lemma 7. The number of nonempty bins ever tested is at most dn, for some constant $d > 0$, by Lemma 8, with the suitably large probability. Applying the union bound results in the estimate n^{-a} on the probability of error for sufficiently large β.

The duration of an iteration of the inner for-loop is either constant, then we call is *short*, or it takes time $\mathcal{O}(\log \texttt{size})$, then we call it *long*. First, we estimate the total time spent on short iterations. This time in the first iteration of the inner for-loop is proportional to `number-of-bins` returned by procedure ESTIMATE-SIZE, which is at most $6n \cdot \lg(6n)$, by Lemma 7. Each of the subsequent iterations takes time proportional to `size`, which is at most $6n$, again by Lemma 7. We obtain that the total number of short iterations is $\mathcal{O}(n \log n)$ in the worst case. Next, we estimate the total time spent on long iterations. One such an iteration has time proportional to $\lg \texttt{size}$, which is at most $\lg 6n$ with certainty. The number of such iterations is at most dn with probability $1 - n^{-\Omega(\log n)}$, for some constant $d > 0$, by Lemma 8. We obtain that the total number of long iterations is $\mathcal{O}(n \log n)$, with the correspondingly large probability. Combining the estimates for short and long iterations, we obtain $\mathcal{O}(n \log n)$ as a bound on time of one iteration of the main repeat-loop. One such an iteration suffices with probability $1 - n^{-\Omega(\log n)}$, by Lemma 8.

Throwing one ball uses $\mathcal{O}(\log n)$ random bits, by Lemma 7. The number of throws is $\mathcal{O}(n)$ with the suitably large probability, by Lemma 8. ◀

Algorithm COMMON-BOUNDED-MC is optimal with respect to the following performance metrics: the expected time $\mathcal{O}(n \log n)$, by Theorem 2, the number of random bits $\mathcal{O}(n \log n)$, by Proposition 2, and the probability of error $n^{-\mathcal{O}(1)}$, by Proposition 3.

6 Common with Unbounded Memory

We consider naming on a Common PRAM in the case when the amount of shared memory is unbounded. The algorithm we propose, called COMMON-UNBOUNDED-MC, is similar to algorithm COMMON-BOUNDED-MC in Section 5, in that it involves a randomized experiment to estimate the number of processors of the PRAM. Such an experiment is then followed by repeatedly throwing balls into bins, testing for collisions, and throwing again if a collision is detected, until eventually no collisions are detected.

Procedure GAUGE-SIZE-MC

```
k ← 1
repeat
    k ← r(k)
    bin_v ← random integer in [1, 2^k]
until the number of selected values of variable bin is ≤ 2^k/β
return ( ⌈2^{k+1}/β⌉ )
```

Figure 7 A pseudocode for a processor v of a Common PRAM, where the number of shared memory cells is unbounded. The constant $\beta > 0$ is the same parameter as in Figure 8, and an increasing function $r(k)$ is also a parameter.

Algorithm COMMON-UNBOUNDED-MC has its pseudocode given in Figure 8. The algorithm is structured as a repeat loop. An iteration starts by invoking procedure GAUGE-SIZE, whose pseudocode is in Figure 7. This procedure returns `size` as an estimate of the number of processors n. Next, a processor chooses randomly a bin in the range $[1, 3\texttt{size}]$. Then it keeps verifying for collisions $\beta \lg \texttt{size}$, in such a manner that when a collision is detected then a new bin is selected form the same range. After such $\beta \lg \texttt{size}$ verifications and possible new selections of bins, another $\beta \lg \texttt{size}$ verifications follow, but without changing the selected bins. When no collision is detected in the second segment of $\beta \lg \texttt{size}$ verifications, then this terminates the repeat-loop, which triggers assigning each station the rank of the selected bin, by a prefix-like computation. If a collision is detected in the second segment of $\beta \lg \texttt{size}$ verifications, then this starts another iteration of the main repeat-loop.

Procedure GAUGE-SIZE-MC returns an estimate of the number n of processors in the form 2^k, for some positive integer k. It operates by trying various values of k, and, for a considered k, by throwing n balls into 2^k bins and next counting how many bins contain balls. Such counting is performed by a prefix-like computation, whose pseudocode is omitted in Figure 7. The additional parameter $\beta > 0$ is a number that affects the probability of underestimating n.

The way in which selections of numbers k is performed is controlled by function $r(k)$, which is a parameter. We will consider two instantiations of this function: one is function $r(k) = k + 1$ and the other is function $r(k) = 2k$.

▶ **Lemma 10.** *If $r(k) = k + 1$ then the value of `size` as returned by* GAUGE-SIZE-MC *satisfies $\texttt{size} \leq 2n$ with certainty and the inequality $\texttt{size} \geq n$ holds with probability $1 - \beta^{-n/3}$.*

If $r(k) = 2k$ then the value of `size` as returned by GAUGE-SIZE-MC *satisfies $\texttt{size} \leq 2\beta n^2$ with certainty and $\texttt{size} \geq \beta n^2/2$ with probability $1 - \beta^{-n/3}$.*

The following Theorem 11 summarizes the performance of algorithm COMMON-UNBOUNDED-MC (see the pseudocode in Figure 8) as a Monte Carlo one. Its proof relies on mapping an execution of the β-process with verifications on executions of algorithm COMMON-UNBOUNDED-MC in a natural manner.

▶ **Theorem 11.** *Algorithm* COMMON-UNBOUNDED-MC *terminates almost surely, for a sufficiently large β. For each $a > 0$, there exists $\beta > 0$ and $c > 0$ such that the algorithm assigns unique names and has the following additional properties with probability $1 - n^{-a}$.*

Algorithm COMMON-UNBOUNDED-MC

```
repeat
   size ← GAUGE-SIZE
   bin_v ← random integer in [1, 3 size]
   for i ← 1 to β lg size do
      if VERIFY-COLLISION (bin_v) then
         bin_v ← random number in [1, 3 size]
   Collision-Detected ← false
   for i ← 1 to β lg size do
      if VERIFY-COLLISION (bin_v) then
         Collision-Detected ← true
until not Collision-Detected
name_v ← the rank of bin_v among selected bins
```

Figure 8 A pseudocode for a processor v of a Common PRAM, where the number of shared memory cells is unbounded. The constant $\beta > 0$ is a parameter impacting the probability of error. The private variable `name` stores the acquired name.

If $r(k) = k + 1$ then at most cn memory cells are ever needed, $cn \ln^2 n$ random bits are ever generated, and the algorithm terminates in time $\mathcal{O}(\log^2 n)$. If $r(k) = 2k$ then at most cn^2 memory cells are ever needed, $cn \ln n$ random bits are ever generated, and the algorithm terminates in time $\mathcal{O}(\log n)$.

The instantiations of algorithm COMMON-UNBOUNDED-MC are close to optimality with respect to some of the performance metrics we consider, depending on whether $r(k) = k + 1$ or $r(k) = 2k$. If $r(k) = k + 1$ then the algorithm's use of shared memory would be optimal if its time were $\mathcal{O}(\log n)$, by Theorem 3, but it misses space optimality by at most a logarithmic factor, since the algorithm's time is $\mathcal{O}(\log^2 n)$. Similarly, for this case of $r(k) = k + 1$, the number of random bits ever generated $\mathcal{O}(n \log^2 n)$ misses optimality by at most a logarithmic factor, by Proposition 2. In the other case of $r(k) = 2k$, the expected time $\mathcal{O}(\log n)$ is optimal, by Theorem 4, the expected number of random bits $\mathcal{O}(n \log n)$ is optimal, by Proposition 2, and the probability of error $n^{-\mathcal{O}(1)}$ is optimal, by Proposition 3, but the amount of used shared memory misses optimality by at most a polynomial factor, by Theorem 4.

7 Conclusion

We considered four variants of the naming problem for an anonymous PRAM, when the number of processors n is unknown, and developed Monte Carlo naming algorithms for each of them. The two algorithms for a bounded number of shared registers are provably optimal with respect to the following three performance metrics: expected time, expected number of generated random bits and probability of error. It is an open problem to develop Monte Carlo algorithms for Arbitrary and Common PRAMs for the case when the amount of shared memory is unbounded, such that they are simultaneously asymptotically optimal with respect to these same three performance metrics: the expected time, the expected number of generated random bits and the probability of error.

References

1. Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. Tight bounds for asynchronous renaming. *Journal of the ACM*, 61(3):18:1–18:51, 2014.
2. Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the* 24*th International Symposium on Distributed Computing (DISC)*, volume 6343 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2010.
3. Dana Angluin. Local and global properties in networks of processors. In *Proceedings of the* 12*th ACM Symposium on Theory of Computing (STOC)*, pages 82–93, 1980.
4. James Aspnes, Faith Ellen Fich, and Eric Ruppert. Relationships between broadcast and shared memory in reliable anonymous distributed systems. *Distributed Computing*, 18(3):209–219, 2006.
5. James Aspnes, Gauri Shah, and Jatin Shah. Wait-free consensus with infinite arrivals. In *Proceedings of the* 34*th ACM Symposium on Theory of Computing (STOC)*, pages 524–533, 2002.
6. Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.
7. Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2):162–183, 2002.
8. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley, 2nd edition, 2004.
9. Paolo Boldi and Sebastiano Vigna. An effective characterization of computability in anonymous networks. In *Proceedings of the* 15*th International Conference on Distributed Computing (DISC)*, volume 2180 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2001.
10. François Bonnet and Michel Raynal. The price of anonymity: Optimal consensus despite asynchrony, crash, and anonymity. *ACM Transactions on Autonomous and Adaptive Systems*, 6(4):23, 2011.
11. Harry Buhrman, Alessandro Panconesi, Riccardo Silvestri, and Paul Vitanyi. On the importance of having an identity or, is consensus really universal? *Distributed Computing*, 18(3):167–176, 2006.
12. Bogdan S. Chlebus, Gianluca De Marco, and Muhammed Talo. Anonymous processors with synchronous shared memory: Las Vegas algorithms. Submitted.
13. Bogdan S. Chlebus, Gianluca De Marco, and Muhammed Talo. Anonymous processors with synchronous shared memory. *CoRR*, abs/1507.02272, 2015.
14. Bogdan S. Chlebus and Dariusz R. Kowalski. Asynchronous exclusive selection. In *Proceedings of the* 27*th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 375–384, 2008.
15. Ömer Eğecioğlu and Ambuj K. Singh. Naming symmetric processes using shared variables. *Distributed Computing*, 8(1):19–38, 1994.
16. Peter van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. The MIT Press, 1990.
17. Yuval Emek, Jochen Seidel, and Roger Wattenhofer. Computability in anonymous networks: Revocable vs. irrecovable outputs. In *Proceedings of the* 41*st International Colloquium on Automata, Languages, and Programming (ICALP), Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 183–195. Springer, 2014.
18. Faith E. Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, 2003.

19 Rachid Guerraoui and Eric Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.

20 Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

21 Prasad Jayanti and Sam Toueg. Wakeup under read/write atomicity. In *Proceedings of the 4th International Workshop on Distributed Algorithms (WDAG)*, volume 486 of *Lecture Notes in Computer Science*, pages 277–288. Springer, 1990.

22 Shay Kutten, Rafail Ostrovsky, and Boaz Patt-Shamir. The Las-Vegas processor identity problem (How and when to be unique). *Journal of Algorithms*, 37(2):468–494, 2000.

23 Richard J Lipton and Arvin Park. The processor identity problem. *Information Processing Letters*, 36(2):91–94, 1990.

24 Alessandro Panconesi, Marina Papatriantafilou, Philippas Tsigas, and Paul M. B. Vitányi. Randomized naming using wait-free shared variables. *Distributed Computing*, 11(3):113–124, 1998.

25 Eric Ruppert. The anonymous consensus hierarchy and naming problems. In *Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS)*, volume 4878 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2007.

26 Naoshi Sakamoto. Comparison of initial conditions for distributed algorithms on anonymous networks. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 173–179, 1999.