DISTRIBUTED ALGORITHMS FOR NAMING ANONYMOUS PROCESSES

by

MUHAMMED TALO

B.S., Inonu University, Turkey, 2005

M.S., Firat University, Turkey, 2007

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Computer Science and Information Systems

2016

This thesis for the Doctor of Philosophy degree by

Muhammed Talo

has been approved for the

Computer Science and Information Systems Program

by

Bogdan S. Chlebus, Advisor

Ellen Gethner, Chair

Michael Mannino

Burt Simon

Tam Vu

September 8, 2016

Talo, Muhammed (Ph.D., Computer Science and Information Systems)

Distributed Algorithms for Naming Anonymous Processes

Thesis directed by Associate Professor Bogdan S. Chlebus

ABSTRACT

We investigate anonymous processors computing in a synchronous manner and communicating via read-write shared memory. This system is known as a parallel random access machine (PRAM). It is parameterized by a number of processors n and a number of shared memory cells. We consider the problem of assigning unique integer names from the interval $[1, n]$ to all n processors of a PRAM. We develop algorithms for each of the eight specific cases determined by which of the following independent properties hold: (1) concurrently attempting to write distinct values into the same memory cell either is allowed or not, (2) the number of shared variables either is unlimited or it is a constant independent of n, and (3) the number of processors n either is known or it is unknown. Our algorithms terminate almost surely, they are Las Vegas when n is known, they are Monte Carlo when n is not known.We show lower bounds on time, depending on whether the amounts of shared memory are constant or unlimited. In view of these lower bounds, all the Las Vegas algorithms we develop are asymptotically optimal with respect to their expected time, as determined by the available shared memory. Our Monte Carlo algorithms are correct 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. We also consider a communication channel in which the only possible communication mode is transmitting beeps, which reach all the nodes instantaneously. The algorithmic goal is to randomly assign names to the anonymous nodes in such a manner that the names make a contiguous segment of positive integers starting from 1. The algorithms are provably optimal with respect to the expected time $\mathcal{O}(n \log n)$, the number of used random bits $\mathcal{O}(n \log n)$, and the probability of error.

The form and content of this abstract are approved. I recommend its publication.

Approved: Bogdan S. Chlebus

ACKNOWLEDGMENT

First of all, I would like to express my gratitude to my advisor Bogdan S. Chlebus. Without his guidance and mentorship this thesis would have been impossible. Not only did he guide me over the years of my graduate studies, also he always tried to understand me even when I clearly was not making any sense. You have set an example of excellence as a researcher, mentor, and role model.

I want to thank Gianluca De Marco for collaboration on several research projects.

I would like to thank Ellen Gethner and Sarah Mandos for all the support over the years.

My research and work on the thesis have also been sponsored by Turkish Government and supported by Bogdan S. Chlebus through the National Science Foundation. This would not have been possible without their support.

I would also like to thank the faculty and my fellow students at the University of Colorado Denver for creating excellent working conditions.

Finally, I thank my daughters Betul and Aise, my wife Dilek, and my parents for their patience and continuing support.

TABLE OF CONTENTS

TABLES

Table

FIGURES

Figure

1. Introduction

We consider a distributed system in which some n processors communicate using read-write shared memory. It is assumed that operations performed on shared memory occur synchronously, in that executions of algorithms are structured as sequences of globally synchronized rounds. Each processor is an independent random access machine with its own private memory. Such a system is known as a (synchronous) Parallel Random Access Machine (PRAM). We consider the problem of assigning distinct integer names from the interval $[1, n]$ to the processors of a PRAM, when originally the processors do not have distinct identifiers.

The problem to assign unique names to anonymous processes in distributed systems can be considered as a stage in either building such systems or making them fully operational. Correspondingly, this may be categorized as either an architectural challenge or an algorithmic one. For example, tightly synchronized message passing systems are typically considered under the assumption that processors are equipped with unique identifiers from a contiguous segment of integers. This is because such systems impose strong demands on the architecture and the task of assigning identifiers to processors is modest when compared to providing synchrony. Similarly, when synchronous parallel machines are designed, then processors may be identified by how they are attached to the underlying communication network. In contrast to that, PRAM is a virtual model in which processors communicate via shared memory; see an exposition of PRAM as a programming environment given by Keller et al. [62]. This model does not assume any relation between the shared memory and processors that identifies individual processors.

Distributed systems with shared read-write registers are usually considered to be asynchronous. Synchrony in such environments can be added by simulations rather than by a supportive architecture or an underlying communication network. Processes do not need to be hardware nodes, instead, they can be virtual computing agents. When a synchronous PRAM is considered, as obtained by a simulation, then the underlying system architecture does not facilitate identifying processors, and so we do not necessarily expect that processors

are equipped with distinct identifiers in the beginning of a simulation.

We view PRAM as an abstract construct which provides a distributed environment to develop algorithms with multiple agents/processors working concurrently; see Vishkin [89] for a comprehensive exposition of PRAM as a vehicle facilitating parallel programing and harnessing the power of multi-core computer architectures. Assigning names to processors by themselves in a distributed manner is a plausible stage in an algorithmic development of such environments, as it cannot be delegated to the stage of building hardware of a parallel machine.

When processors of a distributed/parallel system are anonymous then the task of assigning a unique identifier to each processor is a key step to make the system fully operational, because names are needed for executing deterministic algorithms. We consider *naming* to be the task of assigning unique integers in the range $[1, n]$ to a given set of n processors as their names. Distributed algorithms assigning names to anonymous processors are called *naming* in this thesis. We assume that anonymous processors do not have any features facilitating identification or distinguishing.

We deal with two kinds of randomized (naming) algorithms, called Monte Carlo and 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 have qualities that depend on whether n is known or not, according to the following simple rule: each algorithm for a known n is Las Vegas while each algorithm for an unknown n is Monte Carlo. Our Monte Carlo algorithms have the probability of error converging to zero with a rate that is polynomial in n. Moreover, when incorrect (duplicate) names are assigned, the set of integers used as names makes a contiguous segment starting at the smallest name 1.

We say that a parameter of an algorithmic problem is *known* when it can be used in a

code of an algorithm. We consider two groups of naming problems for a PRAM, depending on whether the number of processors n is known or not.

Additionally, we consider two categories of naming problems depending on how much shared memory is available. In one case, there is a constant number of memory cells, which means that the amount of memory is independent of n but as large as needed for algorithm design. In the other case, the number of shared memory cells is unbounded, but how much is used depends on an algorithm and n. When there is an unbounded amount of memory then $\mathcal{O}(n)$ memory cells actually suffice for the algorithms we develop. We also categorize naming problems depending on whether it is an Arbitrary PRAM (distinct values may be concurrently attempted to be written into a register, an arbitrary one of them gets written) or a Common PRAM variant (only equal values may be concurrently attempted to be written into a register).

Next, we investigate anonymous channel with beeping. There are some n stations attached to the channel that are devoid of any identifiers. Communication proceeds in synchronous rounds. All the stations start together in the same round. The channel provides a binary feedback to all the attached stations: when no stations transmit then nothing is sensed on the communication medium, and when some station does transmit then every station detects a beep.

A beeping channel resembles multiple-access channels, in that it can be interpreted as a single-hop radio network. The difference between the two models is in the feedback provided by each kind of channel. The traditional multiple access channel with collision detection provides the following ternary feedback: silence occurs when no station transmits, a message is heard when exactly one station transmits, and collision is produced by multiple stations transmitting simultaneously, which results in no message heard and can be detected by carrier sensing as distinct from silence. Multiple access channels also come in a variant without collision detection. In such channels the binary feedback is as follows: when exactly one station transmits then the transmitted message is heard by every station, and otherwise,

when either no station or multiple stations transmit, then this results in silence. A channel with beeping has its communication capabilities restricted only to carrier sensing, without even the functionality of transmitting specific bits as messages. The only apparent mode of exchanging information on such a synchronous channel with beeping is to suitably encode it by sequences of beeps and silences.

Modeling communication by a mechanism as limited as beeping has been motivated by diverse aspects of communication and distributed computing. Beeping provides a detection of collision on a transmitting medium by sensing it. Communication by only carrier sensing can be placed in a general context of investigating wireless communication on the physical level and modeling interference of concurrent transmissions, of which the signal-to-interference-plus-noise ratio (SINR) model is among the most popular and well studied; see [54, 61, 85]. Beeping is then a very limited mode of wireless communication, with feedback in the form of either interference or lack thereof. Another motivation comes from biological systems, in which agents exchange information in a distributed manner, while the environment severely restricts how such agents communicate; see [2, 78, 86]. Finally, communication with beeps belongs to the area of distributed computing by weak devices, where the involved agents have restricted computational and communication capabilities. In this context, the devices are modeled as finite-state machines that communicate asynchronously by exchanging states or messages from a finite alphabet. Examples of this approach include the "population-protocols" model introduced by Angluin et at. [7], (see also [9, 11, 73]), and the "stone-age" distributed computing model proposed by Emek and Wattenhoffer [41].

2. The Summary of the Results

We consider randomized algorithms executed by anonymous processors that operate in a synchronous manner using read-write shared memory with a goal to assign unique names to the processors. This problem is investigated in eight specific cases, depending on additional assumptions, and we give an algorithm for each case. The three independent assumptions regard the following: (1) the knowledge of n, (2) the amount of shared memory, and (3) the PRAM variant.

Las Vegas algorithms have been submitted a journal paper are taken from [26]. The naming algorithms we give terminate with probability 1. These algorithms are Las Vegas for a known number of processors n and otherwise they are Monte Carlo. All our algorithms use the optimum expected number $\mathcal{O}(n \log n)$ of random bits. We show that naming algorithms with n processors and $C > 0$ shared memory cells need to operate in $\Omega(n/C)$ expected time on an Arbitrary PRAM, and in $\Omega(n \log n/C)$ expected time on a Common PRAM. We show that any naming algorithm needs to work in the expected time $\Omega(\log n)$; this bound matters when there is an unbounded supply of shared memory. Based on these facts, all our Las Vegas algorithms for the case of known n operate in the asymptotically optimum time, and when the amount of memory is unlimited, they use only an expected amount of space that is provably necessary. The list of the naming problems' specifications and the corresponding Las Vegas algorithms with their performance bounds is summarized in Table 2.1.

Monte Carlo algorithms have been submitted a journal paper are taken from [27]. 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 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 Monte Carlo algorithms with their performance bounds are summarized in Table 2.2.

A Las Vegas and a Monte Carlo naming algorithms for a beeping channel have been submitted a journal paper are taken from [28]. We considered assigning names to anonymous

Table 2.1: Four naming problems, as determined by the PRAM model and the available amount of shared memory, with the respective performance bounds of their solutions. When the number of shared memory cells is not a constant then the given usage is the expected number of shared memory cells that are actually used.

PRAM Model	Memory	Time	Algorithm
Arbitrary	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Arbitrary-Constant-LV in Section 6.1
Arbitrary	$\mathcal{O}(n/\log n)$	$\mathcal{O}(\log n)$	Arbitrary-Unbounded-LV in Section 6.2
Common	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$	Common-Constant-LV in Section 6.3
Common	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	Common-Unbounded-LV in Section 6.4

Table 2.2: 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 only by at most a logarithmic factor.

PRAM Model	Memory	Time	Algorithm
Arbitrary	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Arbitrary-Bounded-MC in Section 7.1
Arbitrary	unbounded	polylog	Arbitrary-Unbounded-MC in Section 7.2
Common	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$	Common-Bounded-MC in Section 7.3
Common	unbounded	polylog	Common-Unbounded-MC in Section 7.4

stations attached to a channel that allows only beeps to be heard. We present a Las Vegas naming algorithm and a Monte Carlo algorithm and show that algorithms are provably optimal with respect to the number of used random bits O(n log n), the expected time O(n log n), and the probability of error.

3. Previous and Related Work

Here we survey the previous work on anonymous naming.

Lipton and Park [70] were the first to consider the naming problem in asynchronous shared memory systems. They studied naming in asynchronous distributed systems with read-write shared memory controlled by adaptive schedulers; they proposed a solution that terminates with positive probability, which can be made arbitrarily close to 1 assuming a known n. They developed a randomized algorithm that solves naming problem. Their algorithm is not guaranteed to terminate; however, if it terminates no two processors will obtain the same names. Once the processors terminated the given names comprise completely the set $\{1, 2, ..., n\}$.

Their algorithm operates as follows. In the beginning of an execution, all processors initialize the contents of shared registers to zeros. Every processor randomly selects an integer i from $\{1, 2, ..., n^2\}$, where n is the number of processors. Then each processor writes 1 to selected cell i. All processors repeat this procedure until there is at least one row which contains n, 1's bit value. Finally, the integer chosen by each processor is used as a name. The algorithm presented in [70], uses $\mathcal{O}(Ln^2)$ bits and terminates $\mathcal{O}(Ln^2)$ time with probability $1 - c^L$ for some constant $c > 1$.

Teng [87] provided a randomized two layer solution for the naming problem considering the same setting as Lipton and Park, (asynchronous processor, the algorithm would work regardless of initial content of shared memory), but his solution improved the failure probability and decreased the space to $\mathcal{O}(n \log^2 n)$ shared bits, with probability at least $1 - \frac{1}{n^c}$, for a constant c. The algorithm terminates $\mathcal{O}(n \log^2 n)$ time.

The author developed a simple algorithm for asynchronous systems with known n, that is a trivial modification of Lipton and Park's algorithm [70]. Teng assumed that n processors divided into K groups with high probability, that is, each group has about n/K processors. Hence, he reduced the problem size from n to n/K. Then he used the similar technique of Lipton and Park for a smaller size problem. The number of processors is unknown in

each group. Therefore, every processor checks if the sum of the number of 1's equals to n in maximum size rows of each group. The author also observed that if n is unknown then no algorithm guaranteed to terminate with correct names.

Lim and Park [69] showed that the naming problem can be solved in $\mathcal{O}(n)$ space; however, they used word operations instead of bit operations. The authors used a shared memory array indexed 1 through n, where n is the number of processors. The basic idea of their algorithm is, at the beginning of the execution, all processors initialize the contents of shared registers to zeros. Then every processor randomly chooses a key and an ID, which is corresponding to the index of a cell, and tries to store its key to claimed cell. When there is more than one processors choose the same cell to write its ID, the processor with the maximum key keeps the claimed ID and the rest of the processors with small keys claim a new ID. If there are no zero entries in the array, i.e., each processor confirms its own ID, then the algorithm terminates. Note that their protocol can fail when more than one processor chooses the same ID and the same key. Additionally, the authors answered the open questions in which Teng posted at the conclusion of [87].

Eğecioğlu and Singh [39] proposed a synchronous algorithm that each processor repeatedly chooses a new random index value, which selection made independently and uniformly at random, and sets the corresponding shared register to 1. Then it counts the number of ones. If the total number of ones equals to n then the processor exits the loop and assign itself ID of the index value. The expected termination time for the synchronous algorithm is $\mathcal{O}(n^2)$. They also proposed a polynomial-time Las Vegas naming algorithm for asynchronous systems with oblivious scheduling of events for a known n under weak shared read-write memory system. Intuitively, the idea in their algorithm is as follows. It uses K copies of an array. Each processor chooses a random index value for each copies of array K rather than a single array and sets the selected register to 1. Then every processor reads all K arrays. Processors perform write operation to each K copy of arrays in ascending order, but afterwards, read all arrays in descending order. A processor keeps executing the algorithm until the total number

of ones in a row equals to the number of processors. Processors may exit from the execution after detecting a successful read because of asynchrony. The processor that exit from the execution after a successful scan records the IDs of the succeeded row to its private memory. Because of asynchrony, the rest of the processors cannot be able to execute a successful scan at the same time. If there is a successful read and the number of detected processors are less than K, the rest of processors repeat the same sequence of steps on a different array, with argument $n-1$ and so on. When a processor exits from the execution after successful read waits for the rest of the processors to exit and then they assign names to themselves in a range of $[1, n]$. The authors also showed that symmetry cannot be broken if the exact number of processes is unknown. Moreover, they observed that the participation of every processor is necessary in order to terminate.

Kutten et al. [68] considered the 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. Authors provided a nonterminating dynamic algorithm, where processes may stop and start taking steps during the execution and then added a static termination detection mechanism which works when the number of processors n is known.

Their dynamic algorithm operates as follows. When a process is active, it randomly selects an ID and always check to see if the same ID claimed by any other processes. A process repeatedly either reads the claimed register or writes a random bit which is chosen independently and randomly and records the chosen value. When a process reads a register, it checks out if the value of register has changed since it has written to it the last time. If the process observes that his claimed ID is also selected by any other processes, it selects randomly a new ID to claim. Note that in their algorithm each process detects the collision in constant time. If a process observes no change after reading the contents of the shared register then it moves on the next iteration. Next, they provided an algorithm to make the

dynamic algorithm to terminate. They assume that all shared registers are initialized. The termination detection algorithm employs a binary tree, where its leaves corresponding to claimed ID. Each process traverses the binary tree from leaves to root by updating the sum of children in each node. When a process sees that the root of the tree has the value n it exits the loop. In other words, the set of claimed IDs is fixed when the root of the binary tree has the value n.

They used the size of $\mathcal{O}(\log n)$ bits for some registers whereas the algorithm of [39, 70, 87] used single bit register. 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, that is to say, there is no terminating algorithm if n is not known or the scheduler is adaptive.

The authors also gave a Las Vegas algorithm which works for unbounded space under any fair scheduler. Finally, they provided a deterministic solution for the naming problem in read-modify-write model by using just one register. This model is a much advanced computational model, where a process can read and update a shared variable in just one step.

Panconesi et al. [79] gave a randomized wait-free naming algorithm in anonymous systems with processes prone to crashes that communicate by single-writer registers. They assume different processes may address a register with different index number and can read all other shared variables. They gave an algorithm that is based on wait-free implementation of α-Test&SetOnce objects for an adaptive scheduler for the case of known n, which works in the expected running time $\mathcal{O}(n \log n \log \log n)$ bit operation with probability at least $1 - o(1)$ while using a namespace of size $(1 + \epsilon)n$, where $\epsilon > 0$. The model considered in that work assigns unique registers to nameless processes and so has a potential to defy the impossibility of wait-free naming for general multi-writer registers as observed by Kutten et al. [68].

Buhrman et al. [23] 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. [10] 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. [5] gave randomized renaming algorithms that act like naming ones, in that process identifiers are not referred to; for more or renaming see [4, 13, 30]. Aspnes et al. [12] considered solving consensus in anonymous systems with infinitely many processes. Attiya et al. [15] and Jayanti and Toueg [60] studied the impact of initialization of shared registers on solvability of tasks like consensus and wakeup in fault-free anonymous systems. Bonnet et al. [21] considered solvability of consensus in anonymous systems with processes prone to crashes but augmented with failure detectors. Guerraoui and Ruppert [55] 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 [82] studied the impact of anonymity of processes on wait-free computing and mutual implementability of types of shared objects.

Lower bounds on PRAM were given by Fich et al. [43], Cook et al. [31], and Beame [19], among others. A review of lower bounds based on information-theoretic approach is given by Attiya and Ellen [14]. Yao's minimax principle was given by Yao [91]; the book by Motwani and Raghavan [77] gives examples of applications.

The problem of concurrent communication in anonymous networks was first considered by Angluin [6]. 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 [44].

The work about anonymous networks that followed was either on specific network topologies or on problems in general message-passing systems. Most popular specific topologies included that of a ring and hypercube. In particular, the ring topology was investigated

by Attiya et al. [16, 17], Flocchini et al. [45], Diks et al. [38], Itai and Rodeh [58], and Kranakis et al. [65], and the hypercube topology was studied by Kranakis and Krizanc [64] and Kranakis and Santoro [67].

Work on algorithmic problems in anonymous networks of general topologies or anonymous/named agents in anonymous/named networks included the following specific contributions. Afek and Matias [3] and Schieber and Snir [84] considered leader election, finding spanning trees and naming in general anonymous networks. Angluin et al. [8] studied adversarial communication by anonymous agents and Angluin et al. [9] considered self-stabilizing protocols for anonymous asynchronous agents deployed in a network of unknown size. Chalopin et al. [24] studied naming and leader election in asynchronous networks when a node knows the map of the network but its position on the map is unknown. Chlebus et al. [29] investigated anonymous complete networks whose links and nodes are subject to random independent failures in which single fault-free node has to wake up all nodes by propagating a wakeup message through the network. Dereniowski and Pelc [36] considered leader election among anonymous agents in anonymous networks. Dieudonné and Pec [37] studied teams of anonymous mobile agents in networks that execute deterministic algorithm with the goal to convene at one node. Fraigniaud et al. [48] considered naming in anonymous networks with one node distinguished as leader. Gąsieniec et al. [52] investigated anonymous agents pursuing the goal to meet at a node or edge of a ring. Glacet et al. [53] considered leader election in anonymous trees. Kowalski and Malinowski [63] studied named agents meeting in anonymous networks. Kranakis et al. [66] investigated computing boolean functions on anonymous networks. Métivier et al. [72] considered naming anonymous unknown graphs. Michail et al. [74] studied the problems of naming and counting nodes in dynamic anonymous networks. Pelc [80] considered activating an anonymous ad hoc radio network from a single source by a deterministic algorithm. Yamashita and Kameda [90] investigated topological properties of anonymous networks that allow for deterministic solutions for representative algorithmic problems.

General questions of computability in anonymous message-passing systems implemented in networks were studied by Boldi and Vigna [20], Emek et al. [40], and Sakamoto [83].

Next, we review work on problems for Beeping Networks. The model of communication by discrete beeping was introduced by Cornejo and Kuhn [32], who considered a general-topology wireless network in which nodes use only carrier sensing to communicate, and developed algorithms for node coloring. They were inspired by "continuous" beeping studied by Degesys et al. [35] and Motskin et al. [76], and by the implementation of coordination by carrier sensing given by Flury and Wattenhofer [46].

Afek et al. [1] considered the problem to find a maximal independent set of nodes in a distributed manner when the nodes can only beep, under additional assumptions regarding the knowledge of the size of the network, waking up the network by beeps, collision detection among concurrent beeps, and synchrony. Brandes et al. [22] studied the problem of randomly estimating the number of nodes attached to a single-hop beeping network. Czumaj and Davies [34] approached systematically the tasks of deterministic broadcasting, gossiping, and multi-broadcasting on the bit level in general-topology symmetric beeping networks. In a related work, Hounkanli and Pelc [56] studied deterministic broadcasting in asynchronous beeping networks of general topology with various levels of knowledge about the network. Förster et al. [47] considered leader election by deterministic algorithms in general multi-hop networks with beeping. Gilbert and Newport [51] studied the efficiency of leader election in a beeping single-hop channel when nodes are state machines of constant size with a specific precision of randomized state transitions. Huang and Moscibroda [57] considered the problems of identifying subsets of stations connected to a beeping channel and compared their complexity to those on multiple-access channels. Yu et al. [92] considered the problem of constructing a minimum dominating set in networks with beeping.

Networks of nodes communicating by beeping share common features with radio networks with collision detection. Ghaffari and Haeupler [49] gave efficient leader election algorithm by treating collision detection as "beeping" and transmitting messages as bit strings. Their

approach by way of “beep waves” was adopted to broadcasting in networks with beeping by Czumaj and Davies [34]. In a related work, Ghaffari et al. [50] developed randomized broadcasting and multi-broadcasting in radio networks with collision detection.

4. Technical Preliminaries

A synchronous shared-memory system in which some n processors operate concurrently is the assumed model of computation. The essential properties of such systems are as follows: (1) shared memory cells have only reading/writing capabilities, and (2) operations of accessing the shared registers are globally synchronized so that processors work in lockstep.

An execution of an algorithm is structured as a sequence of *rounds* so that each processor performs either a read from or a write to a shared memory cell, along with local computation. We assume that a processor carries out its private computation in a round in a negligible portion of the round. Processors can generate as many private random bits per round as needed; all these random bits generated in an execution are assumed to be independent.

Each shared memory cell is assumed to be initialized to 0 as a default value. This assumption simplifies the exposition, but it can be removed as any algorithm assuming such an initialization can be modified in a relatively straightforward manner to work with dirty memory. A shared memory cell can store any value as needed in algorithms, in particular, integers of magnitude that may depend on n; all our algorithms require a memory cell to store $\mathcal{O}(\log n)$ bits. An invocation of either reading from or writing to a memory location is completed in the round of invocation. This model of computation is referred in the literature as the *Parallel Random Access Machine (PRAM)* [59, 81]. PRAM is usually defined as a model with unlimited number of shared-memory cells, by analogy with the random-access machine (RAM) model. We consider the following two instantiations of the model, determined by the amount of shared memory. In one situation, there is a constant number of shared memory cells, which is independent of the number of processors n but as large as needed in the specific algorithm. In the other case, the number of shared memory cells is unlimited in principle, but the expected number of shared registers accessed in an execution depends on n and is sought to be minimized.

A *concurrent read* occurs when a group of processors read from the same memory cell in the same round; this results in each of these processors obtaining the value stored in the

memory cell at the end of the preceding round. A *concurrent write* occurs when a group of processors invoke a write to the same memory cell in the same round. Without loss of generality, we may assume that a concurrent read of a memory cell and a concurrent write to the same memory cell do not occur simultaneously: this is because we could designate rounds only for reading and only for rounding depending on their parity, thereby slowing the algorithm by a factor of two. A clarification is needed regarding which value gets written to a memory cell in a concurrent write, when multiple distinct values are attempted to be written; such stipulations determine suitable variants of the model. We will consider algorithms for the following two PRAM variants determined by their respective concurrent-write semantics.

Common PRAM is defined by the property that when a group of processors want to write to the same shared memory cell in a round then all the values that any of the processors want to write must be identical, otherwise the operation is illegal. Concurrent attempts to write the same value to a memory cell result in this value getting written in this round.

Arbitrary PRAM allows attempts to write any legitimate values to the same memory cell in the same round. When this occurs, then one of these values gets written, while a selection of this value is arbitrary. All possible selections of values that get written need to be taken into account when arguing about correctness of an algorithm.

We will rely on certain standard algorithms developed for PRAMs, as explained in [59, 81]. One of them is for prefix-type computations. A typical situation in which it is applied occurs when there is an array of m shared memory cells, each memory cell storing either 0 or 1. This may represent an array of bins where 1 stands for a nonempty bin while 0 for an empty bin. Let the rank of a nonempty bin of address x be the number of nonempty bins with addresses smaller than or equal to x. Ranks can be computed in time $\mathcal{O}(\log m)$ by using an auxiliary memory of $\mathcal{O}(m)$ cells, assuming there is at least one processor assigned to a nonempty bin, while other processors do not participate. The bins are associated with the leaves of a binary tree. The processors traverse a binary tree from the leaves to the root and

back to the leaves. When updating information at a node, only the information stored at the parent, the sibling and the children is used. We may observe that the same memory can be used repeatedly when such computation needs to be performed multiple times. A possible approach is to verify if the information at a needed memory cell, representing either a parent, a sibling or a child of a visited node, is fresh or rather stale from previous executions. This could be accomplished in the following three steps by a processor. First, the processor erases a memory cell it needs to read by rewriting its present value by a blank value. Second, the processor writes again the value at node it visits, which may have been erased in the previous step by other processors that need the value. Finally, the processor reads again the memory cell it just erased, to see if it stays erased, which means its contents were stale, or not, which means its contents got rewritten so they are fresh.

Balls into bins. Assigning names to processors can be visualized as throwing balls into bins. Imagine that balls are handled by processors and bins are represented by either memory addresses or rounds in a segment of rounds. Throwing a ball means either writing into some memory address a value that represents a ball or choosing a round from a segment of rounds. A *collision* occurs when two balls end up in the same bin; this means that two processors wrote to the same memory address, not necessarily in the same round, or that they selected the same round. The *rank* of a bin containing a ball is the number of bins with smaller or equal names that contain balls. When each in a group of processors throws a ball and there is no collision then this in principle breaks symmetry in a manner that allows to assign unique names in the group, namely, ranks of selected bins may serve as names.

The following terms refer to the status of a bin in a given round. A bin is called *empty* where there are no balls in it. A bin is *singleton* when it contains a single ball. A bin is *multiple* when there are at least two balls in it. Finally, a bin with at least one ball is *occupied*.

The idea of representing attempts to assign names as throwing balls into bins is quite generic. In particular, it was applied by Eğecioğlu and Singh [39], who proposed a syn-

chronous algorithm that repeatedly throws all balls together into all available bins, the selections of bins for balls made independently and uniformly at random. In their algorithm for n processors, we can use $\gamma \cdot n$ memory cells, where $\gamma > 1$. Let us choose $\gamma = 3$ for the following calculations to be specific. This algorithm has an exponential expected-time performance. To see this, we estimate the probability that each bin is either singleton or empty. Let the balls be thrown one by one. After the first $n/2$ balls are in singleton bins, the probability to hit an empty bin is at most $\frac{2.5n}{3n} = \frac{5}{6}$; we treat this as a success in a Bernoulli trial. The probability of $n/2$ such successes is at most $(\frac{5}{6})^{n/2}$, so the expected time to wait for the algorithm to terminate is at least $\left(\sqrt{\frac{6}{5}}\right)^n$, which is exponential in n.

We consider related processes that could be as fast as $\mathcal{O}(\log n)$ in expected time, while still using only $\mathcal{O}(n)$ shared memory cells, see Section 6.4. The idea is to let balls in singleton bins stay put and only move those that collided with other balls by landing in bins that became thereby multiple. To implement this on a Common PRAM, we need a way to detect collisions, which we explain next.

Collisions among balls. We will use a randomized procedure for Common PRAM to verify if a collision occurs in a bin, say, a bin x, which is executed by each processor that selected bin x. This procedure VERIFY-COLLISION is represented in Figure 4.1. There are two arrays TAILS and HEADS of shared memory cells. Bin x is verified by using memory cells TAILS[x] and HEADS[x]. First, the memory cells TAILS[x] and HEADS[x] are set to false each, and next one of these memory cells is selected randomly and set to true.

Lemma 1 *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 $\frac{1}{2}$.*

Proof: When only one processor executes the procedure, then first the processor sets both `Heads`[x] and `Tails`[x] to false and next only one of them to true. This guarantees that `Heads`[x] and `Tails`[x] store different values and so collision is not detected. When some

Procedure VERIFY-COLLISION (x)

initialize $\texttt{Heads}[x] \leftarrow \texttt{Tails}[x] \leftarrow \texttt{false}$

$\texttt{toss}_v \leftarrow$ outcome of tossing a fair coin

$\texttt{if toss}_v = \text{tails then Tails}[x] \leftarrow \texttt{true else Heads}[x] \leftarrow \texttt{true}$

$\texttt{if Tails}[x] = \texttt{Heads}[x] \texttt{ then return true else return false}$

Figure 4.1: 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.

$m > 1$ processors execute the procedure, then collision is not detected only when either all processors set $\texttt{Heads}[x]$ to true or all processors set $\texttt{Tails}[x]$ to true. This means that the processors generate the same outcome in their coin tosses. This occurs with probability 2^{-m+1}, which is at most $\frac{1}{2}$. □

A beeping channel is related to multiple access channels [25]. It is a network consisting of some n stations connected to a communication medium. We consider synchronous beeping channels, in the sense that an execution of a communication algorithm is partitioned into consecutive rounds. All the stations start an execution together. In each round, a station may either beep or pause. When some station beeps in a round, then each station hears the beep, otherwise all the stations receive silence as feedback. When multiple stations beep together in a round then we call this a *collision*.

We say that a parameter of a communication network is *known* when it can be used in codes of algorithms. The relevant parameter used in this thesis is the number of stations n. We consider two cases, in which either n is known or it is not.

Randomized algorithms use random bits, understood as outcomes of tosses of a fair coin. All different random bits used by our algorithms are considered stochastically independent from each other.

Our naming algorithms have as their goal to assign unique identifiers to the stations,

Procedure DETECT-COLLISION

```
toss_v ← outcome of a random coin toss
if toss_v = heads                                   /* first round */
     then beep else pause
if toss_v = tails                                   /* second round */
     then beep else pause
return (a beep was heard in each of the two rounds)
```

Figure 4.2: A pseudocode for a station v. The procedure takes two rounds to execute. It detect a collision and returns "true" when a beep is heard in each of the rounds, otherwise it does not detect a collision and returns "false."

moreover we want names to be integers in the contiguous range $\{1, 2, \ldots, n\}$, which we denote as $[n]$. The Monte Carlo naming algorithm that we develop has the property that the names it assigns make an interval of integers of the form $[k]$ for $k \leq n$, so that when $k < n$ then there are duplicate identifiers assigned as names, which is the only form of error that can occur.

We will use a procedure to detect collisions, called DETECT-COLLISION, whose pseudocode is in Figure 4.2. The procedure is executed by a group of stations, and they all start their executions simultaneously. The procedure takes two rounds. Each of the participating stations simulates the toss of a fair coin, with the outcomes independent among the participating stations. Depending on the outcome of a toss, a station beeps either in the first or the second of the allocated rounds. A collision is detected only when two consecutive beeps are heard.

Lemma 2 *If k stations perform m time-disjoint calls of procedure* DETECT-COLLISION*, each station participating in exactly one call, then collision is not detected in any of these calls with probability 2^{-k+m}.*

Proof: Consider a call of DETECT-COLLISION performed concurrently by i stations, for

$i \geq 1$. We argue by "deferred decisions." One of these stations tosses a coin and determines its outcome X. The other $i-1$ stations participating concurrently in this call also toss their coins; here we have $i - 1 \geq 0$, so there could be no such a station. The only possibility not to detect a collision is for all of these $i-1$ stations also produce X. This happens with probability 2^{-i+1} in this one call. The probability of producing only `false` during the m calls is the product of these probabilities. When we multiply them out over m instances of the procedure being performed, then the outcome is 2^{-k+m}, because numbers i sum up to k and the number of factors is m. □

Pseudocode conventions and notations. We give pseudocode representations of algorithms, as in Figure 4.1. The conventions of pseudocode are summarized next.

We want that, at any round of an execution, all the processors that have not terminated yet to be at the same line of the pseudocode. In particular, when an instruction is conditional on a statement then a processor that does not meet the condition pauses as long as it would be needed for all the processors that meet the condition complete their instructions, even when there are no such processors.

A pseudocode for a processor refers to a number of variables, both shared and private. We use the following notational conventions to emphasize their relevant properties. Shared variables have names starting with a capital letter, while private variables have names all in small letters. When a variable x is a private variable that may have different values at different processors at the same time, then we denote this variable used by a processor v by x_v. Private variables that have the same value at the same time in all the processors are usually used without subscripts, like variables controlling for-loops.

Each station has its private copy of any among the variables used in the pseudocode. When the values of these copies may vary across the stations, then we add the station's name as a subscript of the variable's name to emphasize that, and otherwise, when all the copies of a variable are kept equal across all the stations then no subscript is used.

An assignment instruction of the form $x \leftarrow y \leftarrow \ldots \leftarrow z \leftarrow \alpha$, where $x, y, \ldots, z$ are

variables and α is a value, means to assign α as the value to be stored in all the listed variables $x, y, \ldots, z$.

We use three notations for logarithms. The notation $\lg x$ stands for the logarithm of x to the base 2. The notation $\ln x$ denotes the natural logarithm of x. When the base of logarithms does not matter then we use $\log x$, like in the asymptotic notation $\mathcal{O}(\log x)$.

Properties of naming algorithms. Naming algorithms in distributed environments involving multi-writer read-write shared memory have to be randomized to break symmetry [6, 18]. An eventual assignment of proper names cannot be a sure event, because, in principle, two processors can generate the same strings of random bits in the course of an execution. We say that an event *is almost sure*, or *occurs almost surely*, when it occurs with probability 1. When n processors generate their private strings of random bits then it is an almost sure event that all these strings are eventually pairwise distinct. Therefore, a most advantageous scenario that we could expect, when a set of n processors is to execute a randomized naming algorithm, is that the algorithm eventually terminates almost surely and that at the moment of termination the output is *correct*, in that the assigned names are without duplicates and fill the whole interval $[1, n]$.

5. Lower bounds and impossibilities

In this section, we show impossibility results to justify methodological approach to naming algorithms we apply, and use lower bounds on performance metrics for such algorithms to argue about the optimality of the algorithms developed in subsequent sections.

5.1 Preliminaries

We start with basic definitions, terminologies, and theorems that are discussed throughout this section.

Lower bounds prove that certain problems cannot be solved efficiently without sufficient resources such as time or space. They also give us an idea about when to stop looking for better solutions. *Impossibility results* show that certain problems cannot be solved under certain assumptions. To understand the nature of naming problem it is necessary to understand lower bounds and impossibility results [14, 44].

The *entropy* [33] is the number of bits on average required to describe the random variable. The *entropy of a random variable* is a lower bound on the average number of bits required to represent the random variable. The entropy of a random variable X with a probability mass function $p(x)$ is defined by

$$H(x) = -\sum_{x} p(x) \lg p(x).$$

Yao's Minimax Principle [91, 77] allows us to prove lower bounds on the performance of Las Vegas and Monte Carlo algorithms. Yao's Minimax Principle says that for an arbitrary chosen input distribution, the expected running time of the optimal deterministic algorithm is a lower bound on the expected running time of the optimal randomized algorithm. Yao's Minimax Principle for Las Vegas randomized algorithms as follows. Let $\mathcal{P}$ be a problem with a finite set $\mathcal{X}$ of inputs and a finite set $\mathcal{A}$ be the set of all possible deterministic algorithms that correctly solve the problem $\mathcal{P}$. Let $cost(X, A)$ be the running time of algorithm A for algorithm $A \in \mathcal{A}$ and input $X \in \mathcal{X}$. Let p be a probability distribution over $\mathcal{X}$ and q over $\mathcal{A}$.

Let X_p be a random input chosen according to p and A_q shows a random algorithm chosen according to q. For all distributions p over $\mathcal{X}$ and q over $\mathcal{A}$,

$$\min_{A \in \mathcal{A}} \mathbb{E}\left[cost(X_p, A)\right] \leq \max_{X \in \mathcal{X}} \mathbb{E}\left[cost(X, A_q)\right].$$

Yao's Minimax Principle for Monte Carlo randomized algorithms state that the expected running time of any Monte Carlo algorithm that errs with probability $\lambda \in [0, \frac{1}{2}]$.

5.2 Lower Bounds for a PRAM

We give algorithms that use the expected number of $\mathcal{O}(n \log n)$ random bits with a large probability. This amount of random information is necessary if an algorithm is to terminate almost surely. The following fact is essentially a folklore, but since we do not know if it was proved anywhere in the literature, we give a proof for completeness' sake. Our arguments resort to the notions of information theory [33].

Proposition 1 *If a randomized naming algorithm is correct with probability p_n, when executed by n anonymous processors, then it requires $\Omega(n \log n)$ random bits with probability at least p_n. In particular, a Las Vegas naming algorithm for n processors uses $\Omega(n \log n)$ random bits almost surely.*

Proof: Let us assign conceptual identifiers to the processors, for the sake of argument. These *unknown identifiers* are known only to an external observer and not to algorithms. The purpose of executing the algorithm is to assign explicit identifiers, which we call *given identifiers*.

Let a processor with an unknown name u_i generate string of bits b_i, for $i = 1, \ldots, n$. A distribution of given identifiers among the n anonymous processors, which results from executing the algorithm, is a random variable X_n with a uniform distribution on the set of all permutations of the unknown identifiers. This is because of symmetry: all processors execute the same code, without explicit private identifiers, and if we rearrange the strings generated bits b_i among the processors u_i, then this results in the corresponding rearrangement of the given names.

The underlying probability space consists of $n!$ elementary events, each determined by an assignment of the given identifiers to the processors identified by the unknown identifiers. It follows that each of these events occurs with probability $1/n!$. The Shannon entropy of the random variable X_n is thus $\lg(n!) = \Theta(n \log n)$. The decision about which assignment of given names is produced is determined by the random bits, as they are the only source of entropy, so the expected number of random bits used by the algorithm needs to be as large as the entropy of the random variable X_n.

The property that all assigned names are distinct and in the interval $[1, n]$ holds with probability p_n. An execution needs to generate a total of $\Omega(n \log n)$ random bits with probability at least p_n, because of the bound on entropy. A Las Vegas algorithm terminates almost surely, and returns correct names upon termination. This means that $p_n = 1$ and so that $\Omega(n \log n)$ random bits are used almost surely. □

We consider two kinds of algorithmic naming problems, as determined by the amount of shared memory. One case is for a constant number of shared memory cells, for which we give an optimal lower bound on time for $\mathcal{O}(1)$ shared memory. The other case is when the number of shared memory cells and their capacity are unbounded, for which we give an "absolute" lower bound on time. We begin with lower bounds that reflect the amount of shared memory.

Intuitively, as processors generate random bits, these bits need to be made common knowledge through some implicit process that assigns explicit names. There is an underlying flow of information spreading knowledge among the processors through the available shared memory. Time is bounded from below by the rate of flow of information and the total amount of bits that need to be shared.

On the technical level, in order to bound the expected time of a randomized algorithm, we apply the Yao's minimax principle [91] to relate this expected time to the distributional expected time complexity. A randomized algorithm whose actions are determined by random bits can be considered as a probability distribution on deterministic algorithms. A determin-

istic algorithm has strings of bits given to processors as their inputs, with some probability distribution on such inputs. The expected time of such a deterministic algorithm, give any specific probability distribution on the inputs, is a lower bound on the expected time of a randomized algorithm.

To make such interpretation of randomized algorithms possible, we consider strings of bits of equal length. With such a restriction on inputs, deterministic algorithm may not be able to assign proper names for some assignments of inputs, for example, when all the inputs are equal. We augment such deterministic algorithms in adding an option for the algorithm to withhold a decision on assignment of names and output "no name" for some processors. This is interpreted as the deterministic algorithm needing longer inputs, for which the given inputs are prefixes, and which for the randomized algorithm means that some processors need to generate more random bits.

Regarding probability distributions for inputs of a given length, it always will be the uniform distribution. This is because we will use an assessment of entropy of such a distribution.

Theorem 1 *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 $1/2$.*

Proof: We consider Las Vegas algorithms in this argument, the Monte Carlo case is similar, the difference is in applying Yao's principle for Monte Carlo algorithms. We interpret a randomized algorithm as a deterministic one working with all possible assignments of random bits as inputs with a uniform mass function on the inputs. The expected time of the deterministic algorithm is a lower bound on the expected time of the randomized algorithm.

There are $n!$ possible assignments of given names to the processors. Each of them occurs with the same probability $1/n!$ when the input bit strings are assigned uniformly at random. Therefore the entropy of name assignments, interpreted as a random variable, is

$\lg n! = \Omega(n \log n)$.

Next we consider executions of such a deterministic algorithm on the inputs with a uniform distribution. We may assume without loss of generality that an execution is structured into the following phases, each consisting of $C + 1$ rounds. In the first round of a phase, each processor either writes into a shared memory cell or pauses. In the following rounds of a phase, every processor learns the current values of each among the C memory cells. This may take C rounds for every processor to scan the whole shared memory, but we do not include this reading overhead as contributing to the lower bound. Instead, since this is a simulation anyway, we conservatively assume that the process of learning all the contents of shared memory cells at the end of a phase is instantaneous and complete.

The Common variant of PRAM requires that if a memory cell is written into concurrently then there is a common value that gets written by all the writers. Such a value needs to be determined by the code and the address of a memory cell. This means that, for each phase and any memory cell, a processor choosing to write into this memory cell knows the common value to be written. By the structure of execution, in which all processors read all the registers after a round of writing, any processor knows what value gets written into each available memory cell in a phase, if any is written into a particular cell. This implies that the contents written into shared memory cells may not convey any new information but are already implicit in the states of the processors represented by their private memories after reading the whole shared memory.

When a processor reads all the shared memory cells in a phase, then the only new information it may learn is the addresses of memory cells into which writes were performed and those into which there were no writes. This makes it possible obtain at most C bits of information per phase, because each register was either written into or not.

There are $\Omega(n \log n)$ bits of information that need to be settled and one phase changes the entropy by at most C bits. It follows that the expected number of phases of the deterministic algorithm is $\Omega(n \log n/C)$. By the Yao's principle, $\Omega(n \log n/C)$ is a lower bound on the

expected time of a randomized algorithm. □

For Arbitrary PRAM, writing can spread information through the written values, because different processes can attempt to write distinct strings of bits. The rate of flow of information is constrained by the fact that when multiple writers attempt to write to the same memory cell then only one of them succeeds, if the values written are distinct. This intuitively means that the size of a group of processors writing to the same register determines how much information the writers learn by subsequent reading. These intuitions are made formal in the proof of the following Theorem 2.

Theorem 2 *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 $1/2$.*

Proof: We consider Las Vegas algorithms in this argument, the Monte Carlo case is similar, the difference is in applying Yao's principle for Monte Carlo algorithms. We again replace a given randomized algorithm by its deterministic version that works on assignments of strings of bits of the same length as inputs, with such inputs assigned uniformly at random to the processors. The goal is to use the property that the expected time of this deterministic algorithm, for a given probability distribution of inputs, is a lower bound on the expected time of the randomized algorithm. Next, we consider executions of this a deterministic algorithm.

Similarly as in the proof of Theorem 1, we observe that there are $n!$ assignments of given names to the processors and each of them occurs with the same probability $1/n!$, when the input bit strings are assigned uniformly at random. The entropy of name assignments is again $\lg n! = \Omega(n \log n)$. The algorithm needs to make the processors learn $\Omega(n \log n)$ bits using the available $C > 0$ shared memory cells.

We may interpret an execution as structured into phases, such that each processor performs at most one write in a phase and then reads all the registers. The time of a phase is

assumed conservatively to be $\mathcal{O}(1)$. Consider a register and a group of processors that attempt to write their values into this register in a phase. The values attempted to be written are represented as strings of bits. If some of these values have 0 and some have 1 at some bit position among the strings, then this bit position may convey one bit of information. The maximum amount of information is provided by a write when the written string of bits facilitates identifying the writer by comparing its written value to the other values attempted to be written concurrently to the same memory cell. It follows that this amount is at most the binary logarithm of the size of this group of processors, so that each memory cell written to in a round contributes at most $\lg n$ bits of information because there may be at most n writers to it. So the maximum number of bits of information learnt by the processors in a phase is $C \lg n$.

Since the entropy of the assignment of names is $\lg n! = \Omega(n \log n)$, the expected number of phases of the deterministic algorithm is $\Omega(n \lg n/(C \lg n)) = \Omega(n/C)$. By the Yao's principle, this is also a lower bound on the expected time of a randomized algorithm. □

Next, we consider "absolute" requirements on time for a PRAM to assign unique names to the n available processors. The generality of the lower bound we give stems from the weakness of assumptions. First, nothing is assumed about the knowledge of n. Second, concurrent writing is not constrained in any way. Third, shared memory cells are unbounded in their number and size. Kutten et al. [68] showed that any Las Vegas naming algorithm for asynchronous read-write shared memory systems has expected time $\Omega(\log n)$ against a certain oblivious schedule.

We show next in Theorem 3 that any Las Vegas naming algorithm has $\Omega(\log n)$ expected time for the synchronous schedule of events. The argument we give is in the spirit of similar arguments applied by Cook et al. [31] and Beame [19]. What these arguments share are a formalization of the notion of flow of information during an execution of an algorithm,combined with a recursive estimate of the rate of this flow.

The relation *processor v knows processor w in round t* is defined recursively as follows.

First, for any processor v, we have that v knows v in any round $t > 0$. Second, if a processor v writes to a shared memory cell R in a round t_1 and a processor w reads from R in a round $t_2 > t_1$ such that there was no other write into this memory cell after t_1 and prior to t_2 then processor w knows in round t_2 each processor that v knows in round t_1. Finally, the relation is the smallest transitive relation that satisfies the two postulates formulated above. This means that it is the smallest relation such that if processor v knows processor w in round t_1 and z knows v in round t_2 such that $t_2 > t_1$ then processor z knows w in round t_2. In particular, the knowledge accumulates with time, in that if a processor v knows processor z in round t_1 and round t_2 is such that $t_2 > t_1$ then v knows z in round t_2 as well.

Lemma 3 *Let $\mathcal{A}$ be a deterministic algorithm that assigns distinct names to the processors, with the possibility that some processors output "no name" for some inputs, when each node has an input string of bits of the same length. When algorithm $\mathcal{A}$ terminates with proper names assigned to all the processors then each processor knows all the other processors.*

Proof: We may assume that $n > 1$ as otherwise one processors knows itself. Let us consider an assignment $\mathcal{I}$ of inputs that results in a proper assignment of distinct names to all the processors when algorithm $\mathcal{A}$ terminates. This implies that all the inputs in the assignment $\mathcal{I}$ are distinct strings of bits, as otherwise some two processors, say, v and w that obtain the same input string of bits would either assign themselves the same name or declare "no name" as output. Suppose that processor v does not know w when v halts for inputs from $\mathcal{I}$. Consider an assignment of inputs $\mathcal{J}$ which is the same as $\mathcal{I}$ for processors different from w and such that the input of w is the same as input for v in $\mathcal{I}$. Then the actions of processor v would be the same with $\mathcal{J}$ as with $\mathcal{I}$, because v is not affected by the input of w, so that v would assign itself the same name with $\mathcal{J}$ as with $\mathcal{I}$. But the actions of processor w would be the same in $\mathcal{J}$ as those of v, because their input strings of bits are identical under $\mathcal{J}$. It follows that w would assign itself the name of v, resulting in duplicate names. □

We will use Lemma 3 to asses running times by estimating the number of interleaved

reads and writes needed for processors to get to know all the processors. The rate of learning such information may depend on time, because we do not restrict the amount of shared memory, unlike in Theorems 1 and 2. Indeed, the rate may increase exponentially, under most conservative estimates.

The following Theorem 3 holds for both Common and Arbitrary PRAMs. The argument used in the proof is general enough not to depend on any specific semantics of writing.

Theorem 3 *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 $1/2$.*

Proof: The argument is for a Las Vegas algorithm, the Monte Carlo case is similar. A randomized algorithm can be interpreted as a probability distribution on a finite set of deterministic algorithms. Such an interpretation works when input strings for a deterministic algorithm are of the same length. We consider all such possible lengths for deterministic algorithms, similarly as in the previous proofs of lower bounds.

Let us consider a deterministic algorithm $\mathcal{A}$, and let inputs be strings of bits of the same length. We may structure an execution of this algorithm $\mathcal{A}$ into *phases* as follows. A phase consists of two rounds. In the first round of a phase, each processor either writes to a shared memory cell or pauses. In the second round of a phase, each processor either reads from a shared memory cell or pauses. Such structuring can be done without loss of generality at the expense of slowing down an execution by a factor of at most 2. Observe that the knowledge in the first round of a phase is the same as in the last round of the preceding phase.

Phases are numbered by consecutively increasing integers, starting from 1. A phase i comprised pairs of rounds $\{2i-1, 2i\}$, for integers $i \geq 1$. In particular, the first phase consists of rounds 1 and 2. We also add phase 0 that represents the knowledge before any reads or writes were performed.

We show the following invariant, for $i \geq 0$: a processor knows at most 2^i processors at

the end of phase i. The proof of this invariant is by induction on i.

The base case is for $i = 0$. The invariant follows from the fact that a processor knows only one processor in phase 0, namely itself, and $2^0 = 1$.

To show the inductive step, suppose the invariant holds for a phase $i \geq 0$ and consider the next phase $i + 1$. A processor v may increase its knowledge by reading in the second round of phase $i+1$. Suppose the read is from a shared memory cell R. The latest write into this memory cell occurred by the first round of phase $i+1$. This means that the processor w that wrote to R by phase $i + 1$, as the last one that did write, knew at most 2^i processors in the round of writing, by the inductive assumption and the fact that what is written in phase $i + 1$ was learnt by the immediately preceding phase i. Moreover, by the semantics of writing, the value written to R by w in that round removed any previous information stored in R. Processor v starts phase $i+1$ knowing at most 2^i processors, and also learns of at most 2^i other processors by reading in phase $i+1$, namely, those values known by the latest writer of the read contents. It follows that processor v knows at most $2^i + 2^i = 2^{i+1}$ processors by the end of phase $i + 1$.

When proper names are assigned by such a deterministic algorithm, then each processor knows every other processor, by Lemma 3. A processor knows every other processor in a phase j such that $2^j \geq n$, by the invariant just proved. Such a phase number j satisfies $j \geq \lg n$, and it takes $2 \lg n$ rounds to complete $\lg n$ phases.

Let us consider inputs strings of bits assigned to processors uniformly at random. We need to estimate the expected running time of an algorithm $\mathcal{A}$ on such inputs. Let us observe that, in the context of interpreting deterministic executions for the sake to apply Yao's principle, terminating executions of $\mathcal{A}$ that do not result in names assigned to all the processors could be pruned from a bound on their expected running time, because such executions are determined by bounded input strings of bits that a randomized algorithm would extend to make them sufficiently long to assign proper names. In other words, from the perspective of randomized algorithms, such prematurely ending executions do not represent

real terminating ones.

The expected time of $\mathcal{A}$, conditional on terminating with proper names assigned, is therefore at least $2 \lg n$. We conclude, by the Yao's principle, that any randomized naming algorithm has $\Omega(\log n)$ expected runtime. □

The three lower bounds on time given in this Section may be applied in two ways. One is to infer optimality of time for a given amount of shared memory used. Another is to infer optimality of shared memory use given a time performance. This is summarized in the following Corollary 1.

Corollary 1 *If the expected time of a naming Las Vegas algorithm is $\mathcal{O}(n)$ on an Arbitrary PRAM with $\mathcal{O}(1)$ shared memory, then this time performance is asymptotically optimal. If the expected time of a naming Las Vegas algorithm is $\mathcal{O}(n \log n)$ on a Common PRAM with $\mathcal{O}(1)$ shared memory, then this time performance is asymptotically optimal. If a Las Vegas naming algorithm operates in time $\mathcal{O}(\log n)$ on an Arbitrary PRAM using $\mathcal{O}(n/\log n)$ shared memory cells, then this amount of shared memory is asymptotically optimal. If a Las Vegas naming algorithm operates in time $\mathcal{O}(\log n)$ on a Common PRAM using $\mathcal{O}(n)$ shared memory cells, then this amount of shared memory is optimal.*

Proof: We verify that the lower bounds match the assumed upper bounds. By Theorem 2, a Las Vegas algorithm operates almost surely in $\Omega(n)$ time on an Arbitrary PRAM when space is $\mathcal{O}(1)$. By Theorem 1, a Las Vegas algorithm operates almost surely in $\Omega(n \log n)$ time on a Common PRAM when space is $\mathcal{O}(1)$. By Theorem 2, a Las Vegas algorithm operates almost surely in $\Omega(\log n)$ time on an Arbitrary PRAM when space is $\mathcal{O}(n/\log n)$. By Theorem 1, a Las Vegas algorithm operates almost surely in $\Omega(\log n)$ time on a Common PRAM when space is $\mathcal{O}(n)$. □

A naming algorithm cannot be Las Vegas when n is unknown, as was observed by Kutten et al. [68] in a more general case of asynchronous computations against an oblivious adversary. We show an analogous fact for synchronous computations.

Proposition 2 *There is no Las Vegas naming algorithm for a PRAM with at least two processors that does not refer to the total number of processors.*

Proof: Let us suppose, to arrive at a contradiction, that such a naming Las Vegas algorithm exists. Consider a system of $n \geq 1$ processors, when n is an arbitrary positive integer, and an execution $\mathcal{E}$ on these n 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 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 $n+1 \geq 2$ 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 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 $n+1$ processors is at least as large as the probability to generate the finite random strings for n processors as in $\mathcal{E}$, and additionally to generate $\alpha_{\mathcal{E}}$ for an extra processor v_2, so this probability is positive. □

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 a 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 a 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 $n^{-c}(1 - n^{-\Omega(1)}) = n^{-\Omega(1)}$. □

5.3 Lower Bounds for a Channel with Beeping

We begin with an observation, formulated as Proposition 4, that if the system is sufficiently symmetric then randomness is necessary to break symmetry. The given argument is standard and is given for completeness sake; see [6, 14, 44].

Proposition 4 *There is no deterministic naming algorithm for a synchronous channel with beeping with at least two stations, in which all stations are anonymous, such that it eventually terminates and assigns proper names.*

Proof: We argue by contradiction. Suppose that there exists a deterministic algorithm that eventually terminates with proper names assigned to the anonymous stations. Let all the stations start initialized to the same initial state. The following invariant is maintained in each round: the internal states of the stations are all equal. We proceed by induction on the round number. The base of induction is satisfied by the assumption about the initialization. For the inductive step, we assume that the stations are in the same state, by the inductive assumption. Then either all of them pause or all of them beep in the next round, so that either all of them hear their own beep or all of them pause and hear silence. This results in the same internal state transition, which shows the inductive step. When the algorithm eventually terminates, then each station assigns to itself the identifier determined by its

state. The identifier is the same in all stations because their states are the same, by the invariant. This violates the desired property of names to be distinct, because there are at least two stations with the same name. □

Proposition 4 justifies developing randomized naming algorithms. We continue with "entropy" arguments; see the book by Cover and Thomas [33] for a systematic exposition of information theory. An execution of a naming algorithm coordinates and translated random bits into names. This same amount of entropy needs to be processed/communicated on the channel, by the Shannon's noiseless coding theorem. An analogue of the following Proposition 5 was stated in Proposition 1 for the model of synchronized processors communicating by reading and writing to shared memory.

Proposition 5 *If a randomized naming algorithm for a channel with beeping is executed by n anonymous stations and is correct with probability p_n then it requires $\Omega(n \log n)$ random bits in total to be generated with probability at least p_n. In particular, a Las Vegas naming algorithm uses $\Omega(n \log n)$ random bits almost surely.*

One round of an execution of a naming algorithm allows the stations that do not transmit to learn at most one bit, because, from the perspective of these stations, a round is either silent or there is a beep. Intuitively, the running time is proportional to the amount of entropy that is needed to assign names. This intuition leads to Proposition 6. In its proof, we combine Shannon's entropy [33] with Yao's principle [91].

Proposition 6 *A randomized naming algorithm for a beeping channel with n stations operates in $\Omega(n \log n)$ expected time, when it is either a Las Vegas algorithm or a Monte Carlo algorithm with the probability of error smaller than $1/2$.*

Proof: We apply the Yao's minimax principle to bound the expected time of a randomized algorithm by the distributional complexity of naming. We consider Las Vegas algorithms first.

A randomized algorithm using strings of random bits generated by stations can be considered as a deterministic algorithm $\mathcal{D}$ on all possible assignments of such (sufficiently long) strings of bits to stations as their inputs. We consider assignments of strings of bits of an equal length with the uniform distribution among all such assignments of strings of the same length. On a given assignment of input strings of bits to stations, the deterministic algorithms either assigns proper names or fails to do so. A failure to assign proper names with some input is interpreted as the randomized algorithm continuing to work with additional random bits, which comes at an extra time cost. This is justified by a combination of two factors. One is that the algorithm is Las Vegas and so it halts almost surely, and with a correct output. Another is that the probability to assign a specific finite sequence as a prefix of a used sequence of random bits is positive. So if starting with a specific string of bits, as a prefix of a possibly longer needed string, would mean inability to terminate with a positive probability, then the naming algorithm would not be Las Vegas.

The common length of these input strings is a parameter, and we consider all sufficiently large positive integer values for this parameter such that their exist strings of random bits of this length resulting in assignments of proper names. For a given length of input strings, we remove input assignments that do not result in assignment proper names and consider a uniform distribution of the remaining inputs. This is the same as the uniform distribution conditional on the algorithm terminating with input strings of bits of a given length.

Let us consider such a deterministic algorithm $\mathcal{D}$ assigning names, and using strings of bits at stations as inputs, these strings being of a fixed length, assigned under a uniform distribution for this length, and such that they result in termination. An execution of this algorithm produces a finite binary sequence of bits, where we translate the feedback from the channel round by round, say, with symbol 1 representing a beep and symbol 0 representing silence. Each such a sequence is a binary codeword representing a specific assignment of names. These codewords have also a uniform distribution, by the same symmetry argument as used in the proof of Proposition 1. The expected length of a word in this code is the

expected time of algorithm $\mathcal{D}$. The expected time of algorithm $\mathcal{D}$ is therefore at least $\lg n! = \Omega(n \log n)$, by the Shannon's noiseless coding theorem. We conclude that, by the Yao's principle, the original randomized Las Vegas algorithm has expected time that is $\Omega(n \log n)$.

A similar argument, by the Yao's principle, applies to a Monte Carlo algorithm that is incorrect with a constant probability smaller than 1/2. The only difference in the argument is that when a given assignment of input sting bits does not result in a proper assignment of names, then either the algorithm continues to work with more bits for an extra time, or terminates with error. □

Next, we consider facts that hold when the number of stations n is unknown. The following Proposition 7 is about the inevitability of error. Intuitively, when two computing/communicating agents generate the same string of bits, then their actions are the same, and so they get the same name assigned. In other words, we cannot distinguish the case when there is only one such an agent present from cases when at least two of them work in unison.

Proposition 7 *For an unknown number of station n, if a randomized naming algorithm is executed by n anonymous stations, then an execution is incorrect, in that duplicate names are assigned to different stations, with probability that is at least $n^{-\Omega(1)}$, assuming that the algorithm uses $\mathcal{O}(n \log n)$ random bits with probability $1 - n^{-\Omega(1)}$.*

The proof of Proposition 7 given in Proposition 3 is for the model of synchronous distributed computing in which processors communicate among themselves by reading from and writing to shared registers. The same argument applies to a synchronous beeping channel, when we understand actions of stations as either beeping or pausing in a round.

We conclude this section with a fact about impossibility of developing a Las Vegas naming algorithm when the number of stations n is unknown.

Proposition 8 *There is no Las Vegas naming algorithm for a channel with beeping with at least two stations such that it does not refer to the number of stations.*

The proof of Proposition 8 given in Proposition 2 is for the model of synchronous distributed computing in which processors communicate among themselves by reading from and writing to shared registers. The proof given for Proposition 2 is general enough to be directly applicable here as well, as both models are synchronous. Proposition 8 justifies developing Monte Carlo algorithm for unknown n, which we do in Section 8.2.

6. PRAM: Las Vegas Algorithms

We consider naming of anonymous processors of a PRAM when the number of processors n is known. This problem is investigated in four specific cases, depending on the additional assumptions pertaining to the model, and we give an algorithm for each case. The two independent assumptions regard the amount of shared memory (constant versus unbounded) and the PRAM variant (Arbitrary versus Common).

6.1 Arbitrary with Constant Memory

We present a naming algorithm for Arbitrary PRAM in the case when there are a constant number of shared memory cells. It is called Arbitrary-Constant-LV.

During an execution of this algorithm, processors repeatedly write random strings of bits representing integers to a shared memory cell called `Pad`, and next read `Pad` to verify the outcome of writing. A processor v that reads the same value as it attempted to write increments the integer stored in a shared register `Counter` and uses the obtained number as a tentative name, which it stores in a private variable $\texttt{name}_v$. The values of `Counter` could get incremented a total of less than n times, which occurs when some two processors chose the same random integer to write to the register `Pad`. The correctness of the assigned names is verified by the inequality $\texttt{Counter} \geq n$, because `Counter` was initialized to zero. When such a verification fails then this results in another iteration of a series of writes to register `Pad`, otherwise the execution terminates and the value stored at $\texttt{name}_v$ becomes the final name of processor v. Pseudocode for algorithm Arbitrary-Constant-LV is given in Figure 6.1. It refers to a constant $\beta > 0$ which determines the bounded range $[1, n^\beta]$ from which processors select integers to write to the shared register `Pad`.

Balls into bins. The selection of random integers in the range $[1, n^\beta]$ by n processors can be interpreted as throwing n balls into n^β bins, which we call *β-process*. A collision represents two processors assigning themselves the same name. Therefore an execution of the algorithm can be interpreted as performing such ball placements repeatedly until there is no collision.

Lemma 4 *For each $a > 0$ there exists $\beta > 0$ such that when n balls are thrown into n^β bins during the β-process then the probability of a collision is at most n^{-a}.*

Algorithm ARBITRARY-CONSTANT-LV

```
repeat
    initialize Counter ← name_v ← 0
    bin_v ← random integer in [1, n^β]
    for i ← 1 to n do
        if name_v = 0 then
            Pad ← bin_v
            if Pad = bin_v then
                Counter ← Counter + 1
                name_v ← Counter
until Counter = n
```

Figure 6.1: A pseudocode for a processor v of an Arbitrary PRAM, where the number of shared memory cells is a constant independent of n. The variables `Counter` and `Pad` are shared. The private variable `name` stores the acquired name. The constant $\beta > 0$ is parameter to be determined by analysis.

Proof: Consider the balls thrown one by one. When a ball is thrown, then at most n bins are already occupied, so the probability of the ball ending in an occupied bin is at most $n/n^\beta = n^{-\beta+1}$. No collisions occur with probability that is at least

$$\left(1 - \frac{1}{n^{\beta-1}}\right)^n \geq 1 - \frac{n}{n^{\beta-1}} = 1 - n^{-\beta+2} \,, \tag{6.1}$$

by the Bernoulli's inequality. If we take $\beta \geq a + 2$ then just one iteration of the repeat-loop is sufficient with probability that is at least $1 - n^{-a}$. □

Next we summarize the performance of algorithm ARBITRARY-CONSTANT-LV as a Las Vegas algorithm.

Theorem 4 *Algorithm* ARBITRARY-CONSTANT-LV *terminates almost surely and there is no error when it terminates. For any $a > 0$, there exist $\beta > 0$ and $c > 0$ and such that the algorithm terminates within time cn using at most $cn \ln n$ random bits with probability at least $1 - n^{-a}$.*

Proof: The algorithm assigns consecutive names from a continuous interval starting from 1,

by the pseudocode in Figure 6.1. It terminates after n different tentative names have been assigned, by the condition controlling the repeat loop in the pseudocode of Figure 6.1. This means that proper names have been assigned when the algorithm terminates.

We map an execution of the β-process on an execution of algorithm Arbitrary-Constant-LV in a natural manner. Under such an interpretation, Lemma 4 estimates the probability of the event that the n processors select different numbers in the interval $[1, n^\beta]$ as their values to write to `Pad` in one iteration of the repeat-loop. This implies that just one iteration of the repeat-loop is sufficient with the probability that is at least $1 - n^{-a}$. The probability of the event that i iterations are not sufficient to terminate is at most n^{-ia}, which converges to 0 as i increases, so the algorithm terminates almost surely. One iteration of the repeat-loop takes $\mathcal{O}(n)$ rounds and it requires $\mathcal{O}(n \log n)$ random bits. □

Algorithm Arbitrary-Constant-LV is optimal among Las Vegas naming algorithms with respect to its expected running time $\mathcal{O}(n)$, given the amount $\mathcal{O}(1)$ of its available shared memory, by Corollary 1, and the expected number of random bits $\mathcal{O}(n \log n)$, by Proposition 1 in Section 5.2.

6.2 Arbitrary with Unbounded Memory

We give an algorithm for Arbitrary PRAM in the case when there is an unbounded supply of initialized shared memory cells. This algorithm is called Arbitrary-Unbounded-LV.

The algorithm uses two arrays `Bin` and `Counter` of $\frac{n}{\ln n}$ shared memory cells each. An execution proceeds by repeated attempts to assign names. During each such attempt, the processors work to assign tentative names. Next, the number of distinct tentative names is obtained and if the count equals n then the tentative names become final, otherwise another attempt is made. We assume that each such attempt uses a new segment of memory cells `Counter` initialized to 0s; this is only to simplify the exposition and analysis, because this memory can be reset to 0 with a straightforward randomized algorithm which is omitted. An attempt to assign tentative names proceeds by each processor selecting two integers $\texttt{bin}_v$ and $\texttt{label}_v$ uniformly at random, where $\texttt{bin} \in [1, \frac{n}{\ln n}]$ and $\texttt{label} \in [1, n^\beta]$. Next the processors

Algorithm ARBITRARY-UNBOUNDED-LV

```
repeat
    allocate Counter[1, n/ln n]                    /* fresh memory cells initialized to 0s/*
    initialize position_v ← (0, 0)
    bin ← a random integer in [1, n/ln n]
    label ← a random integer in [1, n^β]
    repeat
        initialize All-Named ← true
        if position_v = (0, 0) then
            Bin[bin] ← label
            if Bin[bin] = label then
                Counter[bin] ← Counter[bin] + 1
                position_v ← (bin, Counter[bin])
            else All-Named ← false
    until All-Named                                /* each processor has a tentative name /*
    name_v ← rank of position_v
until n is the maximum name                        /* no duplicates among tentative names /*
```

Figure 6.2: A pseudocode for a processor v of an Arbitrary PRAM, where the number of shared memory cells is unbounded. The variables `Bin` and `Counter` denote arrays of $\frac{n}{\ln n}$ shared memory cells each, the variable `All-Named` is also shared. The private variable `name` stores the acquired name. The constant $\beta > 0$ is a parameter to be determined by analysis.

repeatedly attempt to write `label` into `Bin`[`bin`]. Each such a write is followed by a read and the lucky writer uses `Counter`[`bin`] to create a pair of numbers (`bin`, `Counter`[`bin`]), after first incrementing `Counter`[`bin`], which is called `bin`'s *position* and is stored in variable `position`. After all processors have their positions determined, we define their ranks as follows. To find the *rank* of $\texttt{position}_v$, we arrange all such pairs in lexicographic order, comparing first on `bin` and then on `Counter`[`bin`], and the rank is the position of this entry in the resulting list, where the first entry has position 1, the second 2, and so on. Ranks can be computed using a prefix-type algorithm operating in time $\mathcal{O}(\log n)$. This algorithm first finds for each $\texttt{bin} \in [1, \frac{n}{\ln n}]$ the sum $s(\texttt{bin}) = \sum_{1 \le i < \texttt{bin}} \texttt{Counter}[i]$. Next, each process v with a position $(\texttt{bin}_v, c)$ assigns to itself $s(\texttt{bin}_v) + c$ as its rank. After ranks have been computed,

they are used as tentative names. Pseudocode for algorithm ARBITRARY-UNBOUNDED-LV is given in Figure 6.2.

In the analysis of algorithm ARBITRARY-UNBOUNDED-LV we will refer to the following bound on independent Bernoulli trials. Let S_n be the number of successes in n independent Bernoulli trials, with p as the probability of success. Let $b(i; n, p)$ be the probability of an occurrence of exactly i successes. For $r > np$, the following bound holds

$$\Pr(S_n \geq r) \leq b(r; n, p) \cdot \frac{r(1-p)}{r - np} \ , \tag{6.2}$$

see Feller [42].

Balls into bins. We consider throwing n balls into $\frac{n}{\ln n}$ bins. Each ball has a label assigned randomly from the range $[1, n^\beta]$, for $\beta > 0$. We say that a *labeled collision* occurs when there are two balls with the same labels in the same bin. We refer to this process as *β-process*.

Lemma 5 *For each $a > 0$ there exists $\beta > 0$ and $c > 0$ such that when n balls are labeled with random integers in $[1, n^\beta]$ and next are thrown into $\frac{n}{\ln n}$ bins during the β-process then there are at most $c \ln n$ balls in every bin and no labeled collision occurs with probability $1 - n^{-a}$.*

Proof: We estimate from above the probabilities of the event that there are more than $c \ln n$ balls in some bin and that there is a labeled collision. We show that each of them can be made to be at most $n^{-a}/2$, from which it follows that at least one of these two events occurs with probability at most n^{-a}.

Let p denote the probability of selecting a specific bin when throwing a ball, which is $p = \frac{\ln n}{n}$. When we set $r = c \ln n$, for a sufficiently large $c > 1$, then

$$b(r; n, p) = \binom{n}{c \ln n} \Big(\frac{\ln n}{n}\Big)^{c \ln n} \Big(1 - \frac{\ln n}{n}\Big)^{n - c \ln n} \ . \tag{6.3}$$

Formula (6.3) translates (6.2) into the following bound

$$\Pr(S_n \geq r) \leq \binom{n}{c \ln n} \Big(\frac{\ln n}{n}\Big)^{c \ln n} \Big(1 - \frac{\ln n}{n}\Big)^{n - c \ln n} \cdot \frac{c \ln n (1 - \frac{\ln n}{n})}{c \ln n - \ln n} \ . \tag{6.4}$$

The right-hand side of (6.4) can be estimated by the following upper bound:

$$\begin{aligned}
& \Big(\frac{en}{c\ln n}\Big)^{c\ln n}\Big(\frac{\ln n}{n}\Big)^{c\ln n}\Big(1-\frac{\ln n}{n}\Big)^{n-c\ln n}\cdot\frac{c}{c-1} \\
= \; & \Big(\frac{e}{c}\Big)^{c\ln n}\Big(1-\frac{\ln n}{n}\Big)^{n}\Big(\frac{n}{n-\ln n}\Big)^{c\ln n}\cdot\frac{c}{c-1} \\
\leq \; & n^{c}c^{-c\ln n}e^{-\ln n}\Big(\frac{n}{n-\ln n}\Big)^{c\ln n}\cdot\frac{c}{c-1} \\
\leq \; & n^{-c\ln c+c-1}\,,
\end{aligned}$$

for each sufficiently large $n > 0$. This is because

$$\Big(\frac{n}{n-\ln n}\Big)^{c\ln n} = \Big(1+\frac{\ln n}{n-\ln n}\Big)^{c\ln n} \leq \exp\Big(\frac{c\ln^2 n}{n-\ln n}\Big)\,,$$

which converges to 1. The probability that the number of balls in some bin is greater than $c\ln n$ is therefore at most $n\cdot n^{-c\ln c+c-1} = n^{-c(\ln c-1)}$, by the union bound. This probability can be made smaller than $n^{-a}/2$ for a sufficiently large $c > e$.

The probability of a labeled collision is at most that of a collision when n balls are thrown into n^{β} bins. This probability is at most $n^{-\beta+2}$ by bound (6.1) used in the proof of Lemma 4. This number can be made at most $n^{-a}/2$ for a sufficiently large β. □

Next we summarize the performance of algorithm Arbitrary-Unbounded-LV as a Las Vegas algorithm.

Theorem 5 *Algorithm* Arbitrary-Unbounded-LV *terminates almost surely and there is no error when the algorithm terminates. For any $a > 0$, there exists $\beta > 0$ and $c > 0$ such that the algorithm assigns names within $c\ln n$ time and generates at most $cn\ln n$ random bits with probability at least $1 - n^{-a}$.*

Proof: The algorithm terminates only when n different names have been assigned, which is provided by the condition that controls the main repeat-loop in Figure 6.2. This means that there is no error when the algorithm terminates.

We map executions of the β-process on executions of algorithm Arbitrary-Unbounded-LV in a natural manner. The main repeat-loop ends after an iteration in which each group

of processors that select the same value for variable `bin` next select distinct values for `label`. We interpret the random selections in an execution as throwing n balls into $\frac{n}{\ln n}$ bins, where a number `bin` determines a bin. The number of iterations of the inner repeat-loop equals the maximum number of balls in a bin.

For any $a > 0$, it follows that one iteration of the main repeat-loop suffices with probability at least $1 - n^{-a}$, for a suitable $\beta > 0$, by Lemma 5. It follows that i iterations are executed by termination with probability at most n^{-ia}, so the algorithm terminates almost surely.

Let us take $c > 0$ as in Lemma 5. It follows that an iteration of the main repeat-loop takes at most $c \ln n$ steps and one processor uses at most $c \ln n$ random bits in this one iteration with probability at least $1 - n^{-a}$. □

Algorithm Arbitrary-Unbounded-LV is optimal among Las Vegas naming algorithms with respect to the following performance measures: the expected time $\mathcal{O}(\log n)$, by Theorem 3, the number of shared memory cells $\mathcal{O}(n/\log n)$ used to achieve this running time, by Corollary 1, and the expected number of used random bits $\mathcal{O}(n \log n)$, by Proposition 1 in Section 5.2.

6.3 Common with Constant Memory

Now we consider the case of Common PRAM when the number of available shared memory cells is constant. We propose an algorithm called Common-Constant-LV.

An execution of the algorithm is organized as repeated "attempts" to assign temporary names. During such attempt, each processor without a name chooses uniformly at random an integer in the interval $[1, \texttt{number-of-bins}]$, where `number-of-bins` is a parameter initialized to n; such a selection is interpreted in a probabilistic analysis as throwing a ball into `number-of-bins` many bins. Next, for each $i \in [1, \texttt{number-of-bins}]$, the processors that selected i, if any, verify if they are unique in their selection of i by executing procedure Verify-Collision (given in Figure 4.1 in Section 4) $\beta \ln n$ times, where $\beta > 0$ is a number that is determined by analysis. After no collision has been detected, a processor that selected

Algorithm COMMON-CONSTANT-LV

```
repeat
    initialize number-of-bins ← n ; name_v ← Last-Name ← 0 ;
        no-collision_v ← true
    repeat
        initialize Collision-Detected ← false
        if name_v = 0 then
            bin_v ← random integer in [1, number-of-bins]
            for i ← 1 to number-of-bins do
                for j ← 1 to β ln n do
                  if bin_v = i then
                    if VERIFY-COLLISION then
                        Collision-Detected ← collision_v ← true
                if bin_v = i and not collision_v then
                  Last-Name ← Last-Name + 1
                  name_v ← Last-Name
        if n − Last-Name > β ln n
            then number-of-bins ← (n − Last-Name)
            else number-of-bins ← n/(β ln n)
    until not Collision-Detected
until Last-Name = n
```

Figure 6.3: A pseudocode for a processor v of a Common PRAM, where there is a constant number of shared memory cells. Procedure VERIFY-COLLISION has its pseudocode in Figure 4.1; lack of parameter means the default parameter 1. The variables `Collision-Detected` and `Last-Name` are shared. The private variable `name` stores the acquired name. The constant β is a parameter to be determined by analysis.

i assigns itself a consecutive name by reading and incrementing the shared variable `Last-Name`. It takes up to β `number-of-bins` $\ln n$ verifications for collisions for all integers in [1, `number-of-bins`]. When this is over, the value of variable `number-of-bins` is modified by decrementing it by the number of new names just assigned, when working with the last `number-of-bins`, unless such decrementing would result in a number `number-of-bins` that is at most $\beta \ln n$, in which case variable `number-of-bins` is set to $n/(\beta \ln n)$. An attempt ends when all processors have tentative names assigned. These names become final when there

are a total of n of them, otherwise there are duplicates, so another attempt is performed. A pseudocode for algorithm COMMON-CONSTANT-LV is in Figure 6.3, in which the main repeat loop represents an attempt to assign tentative names to each processor. An iteration of the inner repeat loop during which `number-of-bins` $> n/(\beta \ln n)$ is called *shrinking* and otherwise it is called *restored*.

Balls into bins. As a preparation of analysis of performance of algorithm COMMON-CONSTANT-LV, we consider a related process of repeatedly throwing balls into bins, which we call *β-process*. The β-process proceeds through *stages*, each representing one iteration of the inner repeat-loop in Figure 6.3. A stage results in some balls removed and some transitioning to the next stage, so that eventually no balls remain and the process terminates.

The balls that participate in a stage are called *eligible* for the stage. In the first stage, n balls are eligible and we throw n balls into n bins. Initially, we apply the principle that after all eligible balls have been placed into bins during a stage, the singleton bins along with the balls in them are removed. A stage after which bins are removed is called *shrinking*. There are k bins and k balls in a shrinking stage; we refer to k as the *length* of this stage. Given balls and bins for any stage, we choose a bin uniformly at random and independently for each ball in the beginning of a stage and next place the balls in their selected destinations. The bins that either are empty or multiple in a shrinking stage stay for the next stage. The balls from multiple bins become eligible for the next stage.

This continues until such a shrinking stage after which at most $\beta \ln n$ balls remain. Then we restore bins for a total of $n/(\beta \ln n))$ of them to be used in the following stages, during which we never remove any bin; these stages are called *restored*. In these final restored stages, we keep removing singleton balls at the end of a stage, while balls from multiple bins stay as eligible for the next restored stage. This continues until all balls are removed.

Lemma 6 *For any $a > 0$, there exists $\beta > 0$ such that the sum of lengths of all shrinking stages in the β-process is at most $2en$, where e is the base of natural logarithms, and there are at most $\beta \ln n$ restored stages, both events holding with probability $1 - n^{-a}$, for sufficiently*

large n.

Proof: We consider two cases depending on the kind of analyzed stages. Let $k \leq n$ denote the length of a stage.

In a shrinking stage, we throw k balls into k bins choosing bins independently and uniformly at random. The probability that a ball ends up singleton can be bounded from below as follows:

$$k \cdot \frac{1}{k}\Big(1-\frac{1}{k}\Big)^{k-1} \geq (e^{-\frac{1}{k}-\frac{1}{k^2}})^{k-1} = e^{-\frac{k-1}{k}-\frac{k-1}{k^2}} = e^{-1+\frac{1}{k}-\frac{1}{k}+\frac{1}{k^2}} = \frac{1}{e}\cdot e^{1/k^2} \geq \frac{1}{e}\ ,$$

where we used the inequality $1-x \geq e^{-x-x^2}$, which holds for $0 \leq x \leq \frac{1}{2}$.

Let Z_k be the number of singleton balls after k balls are thrown into k bins. It follows that the expectancy of Z_k satisfies $\mathbb{E}[Z_k] \geq k/e$.

To estimate the deviation of Z_k from its expected value, we use the bounded differences inequality [71, 75]. Let B_j be the bin of ball b_j, for $1 \leq j \leq k$. Then Z_k is of the form $Z_k = h(B_1, \ldots, B_k)$ where h satisfied the Lipschitz condition with constant 2, because moving one ball to a different bin results in changing the value of h by at most 2 with respect to the original value. The bounded-differences inequality specialized to this instance is as follows, for any $d > 0$:

$$\Pr(Z_k \leq \mathbb{E}[Z_k] - d\sqrt{k}) \leq \exp(-d^2/8)\ . \tag{6.5}$$

We use this inequality for $d = \frac{\sqrt{k}}{2e}$. Then (6.5) implies the following bound:

$$\Pr\Big(Z_k \leq \frac{k}{e} - \frac{k}{2e}\Big) = \Pr\Big(Z_k \leq \frac{k}{2e}\Big) \leq \exp\Big(-\frac{1}{8}\cdot\Big(\frac{\sqrt{k}}{2e}\Big)^2\Big) = \exp\Big(-\frac{k}{32e^2}\Big)\ .$$

If we start a shrinking stage with k eligible balls then the number of balls eligible for the next stage is at most

$$\Big(1-\frac{1}{2e}\Big)\cdot k = \frac{2e-1}{2e}\cdot k\ ,$$

with probability at least $1-\exp(-k/32e^2)$. Let us continue shrinking stages as long as the inequality $\frac{k}{32e^2} > 3a\ln n$ holds. We denote this inequality concisely as $k > \beta \ln n$ for

$\beta = 96e^2a$. Then the probability that every shrinking stage results in the size of the pool of eligible balls decreasing by a factor of at least

$$\frac{2e-1}{2e} = \frac{1}{f}$$

is itself at least

$$\left(1 - e^{-3a \ln n}\right)^{\log_f n} \geq 1 - \frac{\log_f n}{n^{-3a}} \geq 1 - n^{-2a} \ ,$$

for sufficiently large n, by Bernoulli's inequality.

If all shrinking stages result in the size of the pool of eligible balls decreasing by a factor of at least $1/f$, then the total number of eligible balls summed over all such stages is at most

$$n \sum_{i \geq 0} f^{-i} = n \cdot \frac{1}{1 - f^{-1}} = 2en \ .$$

In a restored stage, there are at most $\beta \ln n$ eligible balls. A restored stage happens to be the last one when all the balls become single after their placement, which occurs with probability at least

$$\left(\frac{n/(\beta \ln n) - \beta \ln n}{n/(\beta \ln n)}\right)^{\beta \ln n} = \left(1 - \frac{\beta^2 \ln^2 n}{n}\right)^{\beta \ln n} \geq 1 - \frac{\beta^3 \ln^3 n}{n} \ ,$$

by the Bernoulli's inequality. It follows that there are more than $\beta \ln n$ restored stages with probability at most

$$\left(\frac{\beta^3 \ln^3 n}{n}\right)^{\beta \ln n} = n^{-\Omega(\log n)} \ .$$

This bound is at most n^{-2a} for sufficiently large n.

Both events, one about shrinking stages and the other about restored stages, hold with probability at least $1 - 2n^{-2a} \geq 1 - n^{-a}$, for sufficiently large n. □

Next we summarize the performance of algorithm Common-Constant-LV as Las Vegas one. In its proof, we rely on mapping executions of the β-process on executions of algorithm Common-Constant-LV in a natural manner.

Theorem 6 *Algorithm* Common-Constant-LV *terminates almost surely and there is no error when the algorithm terminates. For any $a > 0$ there exist $\beta > 0$ and $c > 0$ such that the*

algorithm terminates within time $cn \ln n$ *using at most* $cn \ln n$ *random bits with probability* $1 - n^{-a}$.

Proof: The condition controlling the main repeat-loop guarantees that an execution terminates only when the assigned names fill the interval $[1, n]$ so they are distinct.

To analyze time performance, we consider the β-process of throwing balls into bins as considered in Lemma 6. Let $\beta_1 > 0$ be the number β specified in this Lemma, as determined by a replaced by $2a$ in its assumptions. This Lemma gives that the sum of all values of K summed over all shrinking stages is at most $2en$ with probability at least $1 - n^{-2a}$.

For a given K and a number $i \in [1, K]$, procedure VERIFY-COLLISION is executed $\beta \ln n$ times, where β is the parameter in Figure 6.3. If there is a collision then it is detected with probability at least $2^{-\beta \ln n}$. We may take $\beta_2 \geq \beta_1$ sufficiently large so that the inequality $2en \cdot 2^{-\beta_2 \ln n} < n^{-2a}$ holds.

The total number of instances of executing VERIFY-COLLISION during an iteration of the main loop, while K is kept equal to $n/(\beta \ln n)$, is at most n. Observe that the inequality $n \cdot 2^{-\beta_2 \ln n} < n^{-2a}$ holds with probability at most $1 - n^{-2a}$ because $n < 2en$.

If β is set in Figure 6.3 to β_2 then one iteration of the outer repeat-loop suffices with probability at least $1 - 2n^{-2a}$, for sufficiently large n. This is because verifications for collisions detect all existing collisions with this probability. Similarly, this one iteration takes $\mathcal{O}(n \log n)$ time with probability that is at least $1 - 2n^{-2a}$, for sufficiently large n. The claimed performance holds therefore with probability at least $1 - n^{-a}$, for sufficiently large n.

There are at least i iterations of the main repeat-loop with probability at most n^{-ia}, so the algorithm terminates almost surely. □

Algorithm COMMON-CONSTANT-LV is optimal among Las Vegas algorithms with respect to the following performance measures: the expected time $\mathcal{O}(n \log n)$, given the amount $\mathcal{O}(1)$ of its available shared memory, by Corollary 1, and the expected number of random bits $\mathcal{O}(n \log n)$, by Proposition 1 in Section 5.2.

6.4 Common with Unbounded Memory

Now we consider the last case when PRAM is of its Common variant and there is an unbounded amount of shared memory. We propose an algorithm called Common-Unbounded-LV. The algorithm invokes procedure Verify-Collision, whose pseudocode is in Figure 4.1.

An execution proceeds as a sequence of "attempts" to assign temporary names. When such attempt results in assigning temporary names without duplicates then these transient names become final. An attempt begins from each processor selecting an integer from the interval $[1, (\beta + 1)n]$ uniformly at random and independently, where β is a parameter such that only $\beta > 1$ is assumed. Next, for $\lg n$ steps, each process executes procedure Verify-Collision(x) where x is the currently selected integer. If a collision is detected then a processor immediately selects another number in $[1, (\beta + 1)n]$ and continues verifying for a collision. After $\lg n$ such steps, the processors count the total number of selections of different integers. If this number equals exactly n then the ranks of the selected integers are assigned as names, otherwise another attempt to find names is repeated. Computing the number of selections and the ranks takes time $\mathcal{O}(\log n)$. In order to amortize this time $\mathcal{O}(\log n)$ by verifications, such a computation of ranks is performed only after $\lg n$ verifications. Here a rank of a selected x is the number of numbers that are at most x that were selected. A pseudocode for algorithm Common-Unbounded-LV is given in Figure 6.4. Subroutines of prefix-type, like computing the number of selects and ranks of selected numbers are not included in this pseudocode.

Balls into bins. We consider auxiliary processes of placing balls into bins that abstracts operations on shared memory as performed by algorithm Common-Unbounded-LV.

The *β-process* is about placing n balls into $(\beta + 1)n$ bins. The process is structured as a sequence of stages. A stage represents an abstraction of one iteration of the inner for-loop in Figure 6.4 performed as if collisions were detected instantaneously and with certainty. When a ball is moved then it is placed in a bin selected uniformly at random, all such selections

Algorithm COMMON-UNBOUNDED-LV

```
x ← random integer in [1, (β + 1)n]                    /* throw a ball into bin x */
repeat
    for i ← 1 to lg n do
        if VERIFY-COLLISION (x) then
            x ← random integer in [1, (β + 1)n]
    number-occupied-bins ← the total number of selected values for x
until number-occupied-bins = n
name_v ← the rank of bin x among nonempty bins
```

Figure 6.4: A pseudocode for a processor v of a Common PRAM, where the number of shared memory cells is unbounded. The constant β is a parameter that satisfies the inequality $\beta > 1$. The private variable `name` stores the acquired name.

independent from one another. The stages are performed as follows. In the first stage, n balls are placed into $(\beta+1)n$ bins. When a bin is singleton in the beginning of a stage then the ball in the bin stays put through the stage. When a bin is multiple in the beginning of a stage, then all the balls in this bin participate actively in this stage: they are removed from the bin and placed in randomly-selected bins. The process terminates after a stage in which all balls reside in singleton bins. It is convenient to visualize a stage as occurring by first removing all balls from multiple bins and then placing the removed balls in randomly selected bins one by one.

We associate the *mimicking walk* to each execution of the β-process. Such a walk is performed on points with integer coordinates on a line. The mimicking walk proceeds through stages, similarly as the ball process. When we are to relocate k balls in a stage of the ball process then this is represented by the mimicking walk starting the corresponding stage at coordinate k. Suppose that we process a ball in a stage and the mimicking walk is at some position i. Placing this ball in an empty bin decreases the number of balls for the next stage; the respective action in the mimicking walk is to decrement its position from i to $i-1$. Placing this ball in an occupied bin increases the number of balls for the next stage; the

respective action in the mimicking walk is to increment its position from i to $i+1$. The mimicking walk gives a conservative estimates on the behavior of the ball process, as we show next.

Lemma 7 *If a stage of the mimicking walk ends at a position k, then the corresponding stage of the ball β-process ends with at most k balls to be relocated into bins in the next stage.*

Proof: The argument is broken into three cases, in which we consider what happens in the ball process and what are the corresponding actions in the mimicking walk. A number of balls in a bin in a stage is meant to be the final number of balls in this bin at the end of the stage.

In the first case, just one ball is placed in a bin that begins the stage as empty. Then this ball will not be relocated in the next stage. This means that the number of balls for the next stage decreases by 1. At the same time, the mimicking walk decrements its position by 1.

In the second case, some $j \geq 1$ balls land in a bin that is singleton at the start of this stage, so this ball was not eligible for the stage. Then the number of balls in the bin becomes $j+1$ and these many balls will need to be relocated in the next stage. Observe that this contributes to incrementing the number of the eligible balls in the next stage by 1, because only the original ball residing in the singleton bin is added to the set of eligible balls, while the other balls participate in both stages. At the same time, the mimicking walk increments its position by 1 j times.

In the third and final case, some $j \geq 2$ balls land in a bin that is empty at the start of this stage. Then this does not contribute to a change in the number of balls eligible for relocation in the next stage, as these j balls participate in both stages. Let us consider these balls as placed in the bin one by one. The first ball makes the mimicking walk decrement its position. The second ball makes the walk increment its position, so that it returns to the original position as at the start of the stage. The following ball placements, if any, result in

the walk incrementing its positions. □

Random walks. Next we consider a random walk which will estimate the behavior of a ball process. One component of estimation is provided by Lemma 7, in that we will interpret a random walk as a mimicking walk for the ball process.

The random walk is represented as movements of a marker placed on the non-negative side of the integer number line. The movements of the marker are by distance 1 and they are independent. The *random β-walk* has the marker's position incremented with probability $\frac{1}{\beta+1}$ and decremented with probability $\frac{\beta}{\beta+1}$. This may be interpreted as a sequence of independent Bernoulli trials, in which $\frac{\beta}{\beta+1}$ is chosen to be the probability of success. We will consider $\beta > 1$, for which $\frac{\beta}{\beta+1} > \frac{1}{\beta+1}$, which means that the probability of success is greater than the probability of failure.

Such a random β-walk proceeds through *stages*, which are defined as follows. The first stage begins at position n. When a stage begins at a position k then it ends after k moves, unless it reaches the zero coordinate in the meantime. The zero point acts as an absorbing barrier, and when the walk's position reaches it then the random walk terminates. This is the only way in which the walk terminates. A stage captures one round of PRAM's computation and the number of moves in a stage represents the number of writes processors perform in a round.

Lemma 8 *For any numbers $a > 0$ and $\beta > 1$, there exists $b > 0$ such that the random β-walk starting at position $n > 0$ terminates within $b \ln n$ stages with all of them comprising $\mathcal{O}(n)$ moves with probability at least $1 - n^{-a}$.*

Proof: Suppose the random walk starts at position $k > 0$ when a stage begins. Let X_k be the number of moves towards 0 and $Y_k = k - X_k$ be the number of moves away from 0 in such a stage. The total distance covered towards 0, which we call *drift*, is

$$L(k) = X_k - Y_k = X_k - (k - X_k) = 2X_k - k \ .$$

The expected value of X_k is $\mathbb{E}[X_k] = \frac{\beta k}{\beta+1} = \mu_k$. The event $X_k < (1-\varepsilon)\mu_k$ holds with

probability at most $\exp(-\frac{\varepsilon^2}{2}\mu_k)$, by the Chernoff bound [75], so that $X_k \geq (1-\varepsilon)\mu_k$ occurs with the respective high probability. We say that such a stage is *conforming* when the event $X_k \geq (1-\varepsilon)\mu_k$ holds.

If a stage is conforming then the following inequality holds:

$$L(k) \geq 2(1-\varepsilon)\frac{\beta k}{\beta+1} - k = \frac{\beta - 2\beta\varepsilon - 1}{\beta+1}k \ .$$

We want the inequality $\frac{\beta-2\beta\varepsilon-1}{\beta+1} > 0$ to hold, which is the case when $\varepsilon < \frac{\beta-1}{2\beta}$. Let us fix such $\varepsilon > 0$. Now the distance from 0 after k steps starting at k is

$$k - L(k) = (1 - \frac{\beta - 2\beta\varepsilon - 1}{\beta+1}) \cdot k = \frac{2(1+\beta\varepsilon)}{\beta+1} \cdot k \ ,$$

where $\frac{2(1+\beta\varepsilon)}{\beta+1} < 1$ for $\varepsilon < \frac{\beta-1}{2\beta}$. Let $\rho = \frac{\beta+1}{2(1+\beta\varepsilon)} > 1$. Consecutive i conforming stages make the distance from 0 decrease by at least a factor ρ^{-i}.

When we start the first stage at position n and the next $\log_\rho n$ stages are conforming then after these many stages the random walk ends up at a position that is close to 0. For our purposes, it suffices that the position is of distance at most $s \ln n$ from 0, for some $s > 0$, because of its impact on probability. Namely, the event that all these stages are conforming and the bound $s \ln n$ on distance from 0 holds, occurs with probability at least

$$1 - \log_\rho n \cdot \exp(-\frac{\varepsilon^2}{2}\frac{\beta}{\beta+1}s\ln n) \geq 1 - \log_\rho n \cdot n^{-\frac{\varepsilon^2}{2}\frac{\beta}{\beta+1}s} \ .$$

Let us choose $s > 0$ such that

$$\log_\rho n \cdot n^{-\frac{\varepsilon^2}{2}\frac{\beta}{\beta+1}s} \leq \frac{1}{2n^a} \ ,$$

for sufficiently large n.

Having fixed s, let us take $t > 0$ such that the distance covered towards 0 is at least $s \ln n$ when starting from $k = t \ln n$ and performing k steps. We interpret these movements as if this was a single conceptual stage for the sake of the argument, but its duration comprises all stages when we start from $s \ln n$ until we terminate at 0. It follows that the conceptual stage comprises at most $t \ln n$ real stages, because a stage takes at least one round.

If this last conceptual stage is conforming then the distance covered towards 0 is bounded by

$$L(k) \geq \frac{\beta - 2\beta\varepsilon - 1}{\beta + 1} \cdot k \ .$$

We want this to be at least $s \ln n$ for $k = t \ln n$, which is equivalent to

$$\frac{\beta - 2\beta\varepsilon - 1}{\beta + 1} \cdot t > s \ .$$

Now it is sufficient to take $t > s \cdot \frac{\beta+1}{\beta-2\beta\varepsilon-1}$. This last conceptual stage is not conforming with probability at most $\exp(-\frac{\varepsilon^2}{2}\frac{\beta}{\beta+1}t \ln n)$. Let us take t that is additionally big enough for the following inequality

$$\exp(-\frac{\varepsilon^2}{2}\frac{\beta}{\beta+1}t \ln n) = n^{-\frac{\varepsilon^2}{2}\frac{\beta}{\beta+1}t} \leq \frac{1}{2n^a}$$

to hold.

Having selected s and t, we can conclude that there are at most $(s+t)\ln n$ stages with probability at least $1 - n^{-a}$.

Now let us consider only the total number of moves to the left X_m and to the right Y_m after m moves in total, when starting at position n. The event $X_m < (1-\varepsilon) \cdot \frac{\beta}{1+\beta} \cdot m$ holds with probability at most $\exp(-\frac{\varepsilon^2}{2}\frac{\beta}{1+\beta} \cdot m)$, by the Chernoff bound [75], so that $X_m \geq m \cdot \frac{(1-\varepsilon)\beta}{1+\beta}$ occurs with the respective high probability $1 - \exp(-\frac{\varepsilon^2}{2}\frac{\beta}{1+\beta} \cdot m)$. At the same time we have that the number of moves away from zero, which we denote Y_m, can be estimated to be

$$Y_m = m - X_m < m - m \cdot \frac{(1-\varepsilon)\beta}{1+\beta} = \frac{1+\varepsilon\beta}{1+\beta} \cdot m \ .$$

This gives an estimate on the corresponding drift:

$$L(m) = X_m - Y_m > \frac{\beta - 2\beta\varepsilon - 1}{\beta + 1} \cdot m \ .$$

We want the inequality $\frac{\beta-2\beta\varepsilon-1}{\beta+1} > 0$ to hold, which is the case when $\varepsilon < \frac{\beta-1}{2\beta}$. The drift is at least n, with the corresponding large probability, when $m = d \cdot n$ for $d = \frac{\beta+1}{\beta-2\beta\varepsilon-1}$. The drift is at least such with probability exponentially close to 1 in n, which is at least $1 - n^{-a}$ for sufficiently large n. □

Lemma 9 *For any numbers $a > 0$ and $\beta > 1$, there exists $b > 0$ such that the β-process starting at position $n > 0$ terminates within $b \ln n$ stages after performing $\mathcal{O}(n)$ ball throws with probability at least $1 - n^{-a}$.*

Proof: We estimate the behavior of the β-process on n balls by the behavior of the random β-walk starting at position n. The justification of the estimation is in two steps. One is the property of mimicking walks given as Lemma 7. The other is provided by Lemma 8 and is justified as follows. The probability of decrementing and incrementing position in the random β-walk are such that they reflect the probabilities of landing in an empty bin or in an occupied bin. Namely, we use the facts that during executing the β-process, there are at most n occupied bins and at least $\beta \cdot n$ empty bins in any round. In the β-process, the probability of landing in an empty bin is at least $\frac{\beta n}{(\beta+1)n} = \frac{\beta}{\beta+1}$, and the probability of landing in an occupied bin is at most $\frac{n}{(\beta+1)n} = \frac{1}{\beta+1}$. This means that the random β-walk is consistent with Lemma 7 in providing estimates on the time of termination of the β-process from above. □

Incorporating verifications. We consider the *random β-walk with verifications*, which is defined as follows. The process proceeds through stages, similarly as the regular random β-walk. For any round of the walk and a position at which the walk is at, we first perform a Bernoulli trial with the probability $\frac{1}{2}$ of success. Such a trial is referred to as a *verification*, which is *positive* when a success occurs otherwise it is *negative*. After a positive verification a movement of the marker occurs as in the regular β-walk, otherwise the walk pauses at the given position for this round.

Lemma 10 *For any numbers $a > 0$ and $\beta > 1$, there exists $b > 0$ such that the random β-walk with verifications starting at position $n > 0$ terminates within $b \ln n$ stages with all of them comprising the total of $\mathcal{O}(n)$ moves with probability at least $1 - n^{-a}$.*

Proof: We provide an extension of the proof of Lemma 8, which states a similar property of regular random β-walks. That proof estimated times of stages and the number of moves.

Suppose the regular random β-walk starts at a position k, so that the stage takes k moves. There is a constant $d < 1$ such that the walk ends at a position at most dk with probability exponential in k.

Moreover, the proof of Lemma 8 is such that all the values of k considered are at least logarithmic in n, which provides at most a polynomial bound on error. A random walk with verifications is slowed down by negative verifications. Observe that a random walk with verifications that is performed $3k$ times undergoes at least k positive verifications with probability exponential in k by the Chernoff bound [75]. This means that the proof of Lemma 8 can be adapted to the case of random walks with verifications almost verbatim, with the modifications contributed by polynomial bounds on error of estimates of the number of positive verifications in stages. □

Next, we consider a *β-process with verifications*, which is defined as follows. The process proceeds through stages, similarly as the regular ball process. The first stage starts with placing n balls into $(\beta + 1)n$ bins. For any following stage, we first go through multiple bins and, for each ball in such a bin, we perform a Bernoulli trial with the probability $\frac{1}{2}$ of success, which we call a *verification*. A success in a trial is referred to as a *positive verification* otherwise it is a *negative* one. If at least one positive verification occurs for a ball in a multiple bin then all the balls in this bin are relocated in this stage to bins selected uniformly at random and independently for each such a ball, otherwise the balls stay put in this bin until the next stage. The β-process terminates when all the balls are singleton.

Lemma 11 *For any numbers $a > 0$ and $\beta > 1$, there exists $b > 0$ such that the β-process with verifications terminates within $b \ln n$ stages with all of them comprising the total of $\mathcal{O}(n)$ ball throws with probability at least $1 - n^{-a}$.*

Proof: The argument proceeds by combining Lemma 7 with Lemma 10, similarly as the proof of Lemma 9 is proved by combining Lemma 7 with Lemma 8. The details follow.

For any execution of a ball process with verifications, we consider a "mimicking random

walk," also with verifications, defined such that when a ball from a multiple bin is handled then the outcome of a random verification for this ball is mapped on a verification for the corresponding random walk. Observe that for a β-process with verifications just one positive verification is sufficient among $j-1$ trials when there are $j>1$ balls in a multiple bin, so a random β-walk with verifications provides an upper bound on time of termination of the β-process with verifications. The probabilities of decrementing and incrementing position in the random β-walk with verifications are such that they reflect the probabilities of landing in an empty bin or in an occupied bin, similarly as without verifications. All this give a consistency of a β-walk with verifications with Lemma 7 in providing estimates on the time of termination of the β-process from above. □

Next we summarize the performance of algorithm Common-Unbounded-LV as Las Vegas one. The proof is based on mapping executions of the β-processes with verifications on executions of algorithm Common-Unbounded-LV in a natural manner.

Theorem 7 *Algorithm* Common-Unbounded-LV *terminates almost surely and when the algorithm terminates then there is no error. For each $a>0$ and any $\beta>1$ in the pseudocode, there exists $c>0$ such that the algorithm assigns proper names within time $c\lg n$ and using at most $cn\lg n$ random bits with probability at least $1-n^{-a}$.*

Proof: The algorithm terminates when there are n different ranks, by the condition controlling the repeat-loop. As ranks are distinct and each in the interval $[1,n]$, each name is unique, so there is no error. The repeat-loop is executed $\mathcal{O}(1)$ times with probability at least $1-n^{-a}$, by Lemma 11. The repeat-loop is performed i times with probability that is at most n^{-ia}, so it converges to 0 with i increasing. It follows that the algorithm terminates almost surely.

An iteration of the repeat-loop in Figure 6.4 takes $\mathcal{O}(\log n)$ steps. This is because of the following two facts. First, it consists of $\lg n$ iterations of the for-loop, each taking $\mathcal{O}(1)$ rounds. Second, it concludes with verifying the until-condition, which is carried out by

counting nonempty bins by a prefix-type computation. It follows that time until termination is $\mathcal{O}(\log n)$ with probability $1 - n^{-a}$.

By Lemma 11, the total number of ball throws is $\mathcal{O}(n)$ with probability $1 - n^{-a}$. Each placement of a ball requires $\mathcal{O}(\log n)$ random bits, so the number of used random bits is $\mathcal{O}(n \log n)$ with the same probability. □

Algorithm Common-Unbounded-LV is optimal among Las Vegas naming algorithms with respect to the following performance measures: the expected time $\mathcal{O}(\log n)$, by Theorem 3, the number of shared memory cells $\mathcal{O}(n)$ used to achieve this running time, by Corollary 1, and the expected number of random bits $\mathcal{O}(n \log n)$, by Proposition 1.

6.5 Conclusion

We considered the naming problem for the anonymous synchronous PRAM when the number of processors n is known. We gave Las Vegas algorithms for four variants of the problem, which are determined by the suitable restrictions on concurrent writing and the amount of shared memory. Each of these algorithms is provably optimal for its case with respect to the natural performance metrics such as expected time (as determined by the amount of shared memory) and expected number of used random bits.

7. PRAM: Monte Carlo Algorithms

We consider naming of anonymous processors of a PRAM when the number of processors n is unknown. They are determined by two independent specifications the naming problems: the amount of shared memory and the PRAM variant.

7.1 Arbitrary with Constant 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.

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 present, in that 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 as a 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

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 7.1: 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.

integers from the current selection range makes a sufficiently small fraction of the size of the used range.

A pseudocode of algorithm ARBITRARY-BOUNDED-MC is given in Figure 7.1. Its structure is determined by the main repeat-loop. Each iteration of the main 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 progress 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 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.

Balls into bins. We consider the following auxiliary *β-process* of throwing balls into bins, for a parameter $\beta > 0$. The process proceeds through stages identified by consecutive positive integers. The ith stage has the number parameter k equal to $k = 2^i$. During a stage, we first throw n balls into the corresponding 2^k bins and next count the number of occupied bins. A stage is last in an execution of the β-process, and so the β-process terminates, when the number of occupied bins is smaller than or equal to $2^{k/\beta}$. We observe that the β-process always terminates. This is because, by its specification, the β-process terminates by the first stage in which the inequality $n \leq 2^{k/\beta}$ holds and n is an upper bound on the number of occupied bins in a stage. The inequality $n \leq 2^{k/\beta}$ is equivalent to $n^\beta \leq 2^k$ and so to $\beta \lg n \leq k$. Since k goes through consecutive powers of 2, we obtain that the number of stages of the β-process with n balls is at most $\lg(\beta \lg n) = \lg \beta + \lg \lg n$.

We say that such a β-process is *correct* when upon termination each ball is in a separate bin, otherwise the process is *incorrect*.

Lemma 12 *For any $a > 0$ there exists $\beta > 0$ such that the β-process is incorrect with probability that is at most n^{-a}, for sufficiently large n.*

Proof: The β-process is incorrect when there are collisions after the last stage. The probability of the intersection of the events "β-process terminates" and "there are collisions" is

bounded from above by the probability of any one of these events. Next we show that, for each pair of k and n, some of these two events occurs with probability that is at most n^{-a}, for a suitable β.

First, we consider the event that the β-process terminates. The probability that there are at most $2^{k/\beta}$ occupied bins is at most

$$\begin{aligned}\binom{2^k}{2^{k/\beta}}\left(\frac{2^{k/\beta}}{2^k}\right)^n &\leq \left(\frac{e2^k}{2^{k/\beta}}\right)^{2^{k/\beta}} 2^{k(\beta^{-1}-1)n} \\ &\leq e^{2^{k/\beta}} \cdot 2^{k(1-\beta^{-1})2^{k/\beta}} \cdot 2^{k(\beta^{-1}-1)n} \\ &\leq e^{2^{k/\beta}} \cdot 2^{k(\beta^{-1}-1)(n-2^{k/\beta})} \ . \end{aligned} \tag{7.1}$$

We estimate from above the natural logarithm of the right-hand side of (7.1). We obtain the following upper bound:

$$\begin{aligned}2^{k/\beta} + k(\beta^{-1}-1)(n-2^{k/\beta})\ln 2 &< 2^{k/\beta} - \frac{1}{2}(n-2^{k/\beta})\ln 2 \\ &= 2^{k/\beta} - \frac{\ln 2}{2}n + \frac{\ln 2}{2}2^{k/\beta} \\ &= -\frac{\ln 2}{2}n + 2^{k/\beta}\cdot\frac{2+\ln 2}{2} \ , \end{aligned} \tag{7.2}$$

for $\beta > 4/3$, as $k \geq 2$. The estimate (7.2) is at most $-n\cdot\frac{\ln 2}{4}$ when $2^{k/\beta} \leq n\cdot\delta$, for $\delta = \frac{\ln 2}{2(2+\ln 2)}$, by a direct algebraic verification. These restrictions on k and β can be restated as

$$k \leq \beta \lg(n\delta) \text{ and } \beta > 4/3 \ . \tag{7.3}$$

When this condition (7.3) is satisfied, then the probability of at most $2^{k/\beta}$ occupied bins is at most

$$\exp\left(-n\cdot\frac{\ln 2}{4}\right) \leq n^{-a}$$

for sufficiently large n.

Next, let us consider the probability of collisions occurring. Collisions do not occur with probability that is at least

$$\left(1-\frac{n}{2^k}\right)^n \geq 1 - \frac{n^2}{2^k} \ ,$$

by the Bernoulli's inequality. It follows that the probability of collisions occurring can be bounded from above by $\frac{n^2}{2^k}$. This bound in turn is at most n^{-a} when

$$k \geq (2+a)\lg n \ . \tag{7.4}$$

In order to have some of the inequalities (7.3) and (7.4) hold for any k and n, it is sufficient to have

$$(2+a)\lg n \leq \beta \lg(n\delta) \ .$$

This determines β as follows:

$$\beta \geq \frac{(2+a)\lg n}{\lg n + \lg \delta} \rightarrow 2+a \ ,$$

with $n \rightarrow \infty$. We obtain that the inequality $\beta > 2+a$ suffices, for n that is large enough. □

Lemma 13 *For each $\beta > 0$ there exists $c > 0$ such that when the β-process terminates then the number of bins ever needed is at most cn and the number of random bits ever generated is at most $cn \ln n$.*

Proof: The β-process terminates by the stage in which the inequality $n \leq 2^{k/\beta}$ holds, so k gets to be at most $\beta \lg n$. We partition the range $[2, \beta \lg n]$ of values of k into two subranges and consider them separately.

First, when k ranges from 2 to $\lg n$ through the stages, then the numbers of needed bins increase quadratically through the stages, because k is doubled with each transition to the next stage. This means that the total number of all these bins is $\mathcal{O}(n)$. At the same time, the number of random bits increases geometrically through the stages, so the total number of random bits a processor uses is $\mathcal{O}(\log n)$.

Second, when k ranges from $\lg n$ to $\beta \lg n$, the number of needed bins is at most n in each stage. There are only $\lg(\beta+1)$ such stages, so the total number of all these bins is $\lg(\beta+1) \cdot n$. At the same time, a processor uses at most $\beta \lg n$ random bits in each of these stages. □

There is a direct correspondence between iterations of the outer repeat-loop and stages of a β-process. The ith stage has the number k equal to the value of k during the ith iteration of the outer repeat-loop of algorithm ARBITRARY-BOUNDED-MC, that is, we have $k = 2^i$. We map an execution of the algorithm into a corresponding execution of a β-process in order to apply Lemmas 12 and 13 in the proof of the following Theorem, which summarizes the performance of algorithm ARBITRARY-BOUNDED-MC and justifies that it is Monte Carlo.

Theorem 8 *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 $1 - n^{-a}$.*

Proof: The number of stages of the β-process with n balls is at most $\lg(\beta \lg n) = \lg \beta + \lg \lg n$. This is also an upper bound on the number of iterations of the main repeat-loop. We conclude that the algorithm always terminates.

The number of bins available in a stage is an upper bound on the number of bins occupied in this stage. The number of bins occupied in a stage equals the number of times the inner repeat-loop is iterated, because executing instruction `Pad` $\leftarrow$ `bin` eliminates one occupied bin. It follows that the number of bins ever needed is an upper bound on time of the algorithm. The number of iterations of the inner repeat-loop is executed is recorded in the variable `Last-Name`, so the termination condition of the algorithm corresponds to the termination condition of the β-process.

When the β-process is correct then this means that the processors obtain distinct names. We conclude that Lemmas 12 and 13 apply when understood about the behavior of the algorithm. This implies the following: the names are correct and execution terminates in $\mathcal{O}(n)$ time while $\mathcal{O}(n \log n)$ bits are used, all this with probability that is 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 2, the expected number of random

bits $\mathcal{O}(n \log n)$, by Proposition 1, and the probability of error $n^{-\mathcal{O}(1)}$, by Proposition 3.

7.2 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.

The underlying idea is to parallelize the process of selection of names applied in Section 7.1 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 uses 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 to processors based on their lexicographic ordering by pairs of numbers determined by `Pad` and `Last-Name`.

A pseudocode for algorithm Arbitrary-Unbounded-MC is given in Figure 7.2. It is structured as a repeat-loop. In the first iteration, the parameter k equals 1, and in subsequent ones is determined by iterations of an 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 `Last-Name`[`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

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 7.2: 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.

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.

Balls into bins. We consider a related process of placing labeled balls into bins, which is referred to as *β-process*. Such a process proceeds through stages and is parametrized by a function $r(k)$. In the first stage, we have $k = 1$, and given some value of k in a stage, the next stage has this parameter equal to $r(k)$. In a stage with a given k, we place n balls into $2^k/(\beta k)$ bins, with labels from $[1, 2^{\beta k}]$. The selections of bins and labels are performed independently and uniformly at random. A stage terminates the β-process when there are

at most βk labels of balls in each bin.

Lemma 14 *The β-process always terminates.*

Proof: The β-process terminates by a stage in which the inequality $n \leq \beta k$ holds, because n is an upper bound on the number of balls in a bin. This always occurs when function $r(k)$ is increasing. □

We expect the β-process to terminate earlier, as the next Lemma states.

Lemma 15 *For each $a > 0$, if $k \leq \lg n - 2$ and $\beta \geq 1 + a$ then the probability of halting in the stage is smaller than n^{-a}, for sufficiently large n.*

Proof: We show that when k is suitably small then the probability of at most βk different labels in each bin is small. There are n balls placed into $2^k/(\beta k)$ bins, so there are at least $\frac{\beta k n}{2^k}$ balls in some bin, by the pigeonhole principle. We consider these balls and their labels.

The probability that all these balls have at most βk labels is at most

$$\begin{aligned}\binom{2^{\beta k}}{\beta k}\Big(\frac{\beta k}{2^{\beta k}}\Big)^{\frac{\beta k n}{2^k}} &\leq \Big(\frac{e 2^{\beta k}}{\beta k}\Big)^{\beta k} \cdot \frac{(\beta k)^{\frac{\beta k n}{2^k}}}{(2^{\beta k})^{\frac{\beta k n}{2^k}}} \\ &= e^{\beta k} 2^{\beta k(\beta k - \frac{\beta k n}{2^k})} (\beta k)^{\frac{\beta k n}{2^k} - \beta k} \\ &= e^{\beta k}\Big(\frac{\beta k}{2^{\beta k}}\Big)^{\frac{\beta k n}{2^k} - \beta k} . \end{aligned} \tag{7.5}$$

We want to show that this is at most n^{-a}. We compare the logarithms (But the base of logarithms!) of n^{-a} and the right-hand side of (7.5), and want the following inequality to hold:

$$\beta k + \Big(\frac{\beta k n}{2^k} - \beta k\Big)(\lg(\beta k) - \beta k) \leq -a \lg n \ ,$$

which is equivalent to the following inequality, by algebra:

$$\frac{n}{2^k} \geq \frac{1}{\beta k - \lg(\beta k)} + 1 + \frac{a \lg n}{\beta k(\beta k - \lg(\beta k))} \ . \tag{7.6}$$

Observe now that, assuming $\beta \geq a + 1$, if $k < \sqrt{\lg n}$ then the right-hand side of (7.6) is at most $2 + \lg n$ while the left-hand side is at least $\sqrt{n}$, and when $\sqrt{\lg n} \leq k \leq \lg n - 2$ then

right-hand side of (7.6) is at most 3 while the left-hand side is at least 4, for sufficiently large n. □

We say that a *label collision* occurs, in a configuration produced by the process, if some bin contains two balls with the same label.

Lemma 16 *For any $a > 0$, if $k > \frac{1}{2}\lg n$ and $\beta > 4a + 7$ then the probability of a label collision is smaller than n^{-a}.*

Proof: The number of pairs of a bin number and a label is $2^k \cdot 2^{\beta k}/(\beta k)$. It follows that the probability of some two balls in the same bin obtaining different labels is at least

$$\left(1 - \frac{n}{2^{k+\beta k}/(\beta k)}\right)^n \geq 1 - \frac{n^2}{2^{k+\beta k}/(\beta k)} \ ,$$

by the Bernoulli's inequality. So the probability that two different balls obtain the same label is at most $\frac{n^2}{2^{k+\beta k}/(\beta k)}$. We want the following inequality to hold

$$\frac{n^2}{2^{k+\beta k}/(\beta k)} < n^{-a} \ .$$

This is equivalent to the inequality obtained by taking logarithms

$$(2 + a)\lg n < (1 + \beta)k - \lg(\beta k) \ ,$$

which holds when $(2 + a)\lg n < \frac{1+\beta}{2}k$. It follows that it is sufficient for k to satisfy

$$k > \frac{2(2 + a)}{1 + \beta}\lg n \ .$$

This inequality holds for $k > \frac{1}{2}\lg n$ when $\beta > 4a + 7$. □

We say that such a β-process is *correct* when upon termination no label collision occurs, otherwise the process is *incorrect*.

Lemma 17 *For any $a > 0$, there exists $\beta > 0$ such that the β-process is incorrect with probability that is at most n^{-a}, for sufficiently large n.*

Proof: The β-process is incorrect when there is a label collision after the last stage. The probability of the intersection of the events "β-process terminates" and "there are label collisions" is bounded from above by the probability of any one of these events. Next we show that, for each pair of k and n, some of these two events occurs with probability that is at most n^{-a}, for a suitable β.

To this end we use Lemmas 15 and 16 in which we substitute $2a$ for a. We obtain that, on the one hand, if $k \leq \lg n - 2$ and $\beta \geq 1 + 2a$ then the probability of halting is smaller than n^{-2a}, and, on the other hand, that if $k > \frac{1}{2}\lg n$ and $\beta > 8a + 7$ then the probability of a label collision is smaller than n^{-2a}. It follows that some of the two considered events occurs with probability at most $2n^{-2a}$ for sufficiently large β and any sufficiently large n. This probability is at most n^{-a}, for sufficiently large n. □

Lemma 18 *For any $a > 0$, there exists $\beta > 0$ and $c > 0$ such that the following two facts about the β-process hold. If $r(k) = k+1$ then at most $cn/\ln n$ bins are ever needed and $cn\ln^2 n$ random bits are ever generated, each among these properties occurring with probability that is at least $1 - n^{-a}$. If $r(k) = 2k$ then at most $cn^2/\ln n$ bins are ever needed and $cn\ln n$ random bits are ever generated, each among these properties occurring with probability that is at least $1 - n^{-a}$.*

Proof: We throw n balls into $2^k/(\beta k)$ bins. As k keeps increasing, then the probability of termination increases as well, because both $2^k/(\beta k)$ and βk increase as functions of k. Let us take $k = 1 + \lg n$ so that the number of bins is $\frac{2n}{\beta k}$. We want to show that no bin contains more than βk balls with a suitably small probability.

Let us consider a specific bin and let X be the number of balls in this bin. The expected number of balls in the bin is $\mu = \frac{\beta k}{2}$. We use the Chernoff bound for a sequence of Bernoulli trials in the form of

$$\Pr(X > (1+\varepsilon)\mu) < \exp(-\varepsilon^2\mu/3) \ ,$$

which holds for $0 < \varepsilon < 1$, see [75]. Let us choose $\varepsilon = \frac{1}{2}$, so that $1 + \varepsilon = \frac{3}{2}$ and $\frac{3}{2}\mu = \frac{3}{4}\beta k$.

We obtain that

$$\Pr(X > \beta k) < \Pr\big(X > \frac{3}{4} \cdot \beta k\big) < \exp(-\frac{1}{4} \cdot \frac{\beta k}{6}) = \exp\big(-\frac{\beta}{24} \cdot (1 + \lg n)\big) \ ,$$

which can be made smaller than n^{-1-a} for a β sufficiently large with respect to a, and sufficiently large n. Using the union bound, each of the n bins contains at most βk balls with probability at most n^{-a}. This implies that termination occurs as soon as k reaches or surpasses $k = 1 + \lg n$, with the corresponding large probability $1 - n^{-a}$.

In the case of $r(k) = k+1$, the consecutive integer values of k are tried, so the β-process terminates by the time $k = 1 + \lg n$, and for this k the number of bins needed is $\Theta(n/\log n)$. To choose a bin for any value of k requires at most k random bits, so implementing such choices for $k = 1, 2, \ldots, 1 + \lg n$ requires $\mathcal{O}(\log^2 n)$ random bits per processor.

In the case of $r(k) = 2k$, the β-process terminates by k equal to $2(1 + \lg n)$, and for this value of k the number of bins needed is $\Theta(n^2/\log n)$. As k progresses through consecutive powers of 2, the sum of these numbers is a sum of a geometric progression, and so is of the order of the maximum term, that is $\Theta(\log n)$, which is the number of random bits per processor. □

There is a direct correspondence between iterations of the outer repeat-loop of algorithm Arbitrary-Unbounded-MC and stages of the β-process. We map an execution of the algorithm into a corresponding execution of a β-process in order to apply Lemmas 17 and 18 in the proof of the following Theorem, which summarizes the performance of algorithm Arbitrary-Unbounded-MC and justifies that it is Monte Carlo.

Theorem 9 *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*

$\mathcal{O}(\log n)$.

Proof: The algorithm always terminates by Lemma 14. By Lemma 17, the algorithm assigns correct names with probability that is at least $1 - n^{-a}$. The remaining properties follow from Lemma 18, because the number of bins is proportional to the number of memory cells and the number of random bits per processor is proportional to time. □

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 2, but it may miss space optimality by at most a logarithmic factor, since the algorithm's 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 1. On the pother hand, if $r(k) = 2k$ then the expected time $\mathcal{O}(\log n)$ is optimal, by Theorem 3, the expected number of random bits $\mathcal{O}(n \log n)$ is optimal, by Proposition 1, 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 2.

7.3 Common with Bounded Memory

Algorithm Common-Bounded-MC solves the naming problem for Common PRAM with a constant number of shared read-write registers. To make its exposition more modular, we use two procedures Estimate-Size and Extend-Names. Procedure Estimate-Size produces 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 algorithm hinges of calibrating the number of bins to the expected number of balls to be thrown.

Algorithm COMMON-BOUNDED-MC has its pseudocode in Figure 7.5. The private variables 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. The pseudocodes of procedures ESTIMATE-SIZE and EXTEND-NAMES are given in Figures 7.3 and 7.4, respectively.

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

Proof: The procedure returns $3 \cdot 2^k$, for some integer $k > 0$. We interpret selecting of values for variable `bin` in an iteration of the main repeat-loop as throwing n balls into $k2^k$ bins; here $k = j + 2$ in the jth iteration of this loop, because the smallest value of k is 3. Clearly, n is an upper bound on the number of occupied bins.

If n is a power of 2, say $n = 2^i$, then the procedure terminates by the time $i = k$, so that $2^k < 2^{i+1} = 2n$. Otherwise, the maximum possible k equals $\lceil \lg n \rceil$, because $2^{\lfloor \lg n \rfloor} < n < 2^{\lceil \lg n \rceil}$. This gives $2^{\lceil \lg n \rceil} = 2^{\lfloor \lg n \rfloor + 1} < 2n$. We obtain that the inequality $2^k < 2n$ occurs with certainty, and so $3 \cdot 2^k < 6n$ does.

Now we estimate the lower bound on 2^k. Consider k such that $2^k \leq \frac{n}{3}$. Then n balls fall

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 7.3: A pseudocode for a processor v of a Common PRAM. This procedure is invoked by algorithm COMMON-BOUNDED-MC in Figure 7.5. The variable `Nonempty-Bins` is shared.

into at most 2^k bins with probability that is at most

$$\binom{k2^k}{2^k}\left(\frac{2^k}{k2^k}\right)^n \leq \left(\frac{ek2^k}{2^k}\right)^{2^k} \cdot \frac{1}{k^n} = (ek)^{2^k} k^{-n} = e^{2^k} k^{2^k - n} \leq e^{n/3} k^{-2n/3} \; . \tag{7.7}$$

The right-hand side of (7.7) is at most $e^{-n/3}$ when the inequality $k > e$ holds. The smallest k considered in the pseudocode in Figure 7.3 is $k = 3 > e$. The inequality $k > e$ is consistent with $2^k \leq \frac{n}{3}$ when $n \geq 20$. The number of possible values for k is $\mathcal{O}(\log n)$ so the probability of the procedure returning for $2^k \leq \frac{n}{3}$ is $e^{-n/3} \cdot \mathcal{O}(\log n) = 2^{-\Omega(n)}$. □

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 7.4 is to be executed. Such a verification is performed by using a shared register initialized to 0,

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 7.4: A pseudocode for a processor v of a Common PRAM. This procedure invokes procedure VERIFY-COLLISION, whose pseudocode is in Figure 4.1, and is itself invoked by algorithm COMMON-BOUNDED-MC in Figure 7.5. 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.

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 7.4 is executed. To prepare for this, 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 7.5, 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 `number-of-bins` $\leftarrow$ `size` is performed. Here, we need to notice that `number-of-bins` $= \Theta(n \log n)$ and `size` $= \Theta(n)$, by the pseudocodes in Figures 7.3 and 7.5 and Lemma 19.

Balls into bins for the second time. 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 7.4, which means that there are n balls and the numbers of bins are as determined by an execution of procedure ESTIMATE-SIZE, that is, the first stage uses `number-of-bins` bins and subsequent stages use `size` bins, as returned by ESTIMATE-SIZE. The only difference from the actions of procedure EXTEND-NAMES is that collisions are detected with certainty in the ball process rather than being tested for, which implies that the parameter β is not involved. The ball process terminates in stage $\lg$ `size` or earlier in the first stage in which no multiple bins are produced, when such a stage occurs.

Lemma 20 *The ball process results in all balls ending singleton 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 $1 - n^{-\Omega(\log n)}$.*

Proof: The argument leverages the property that, in each stage, the number of bins exceeds the number of balls by at least a logarithmic factor. We will denote the number of bins in a stage by m. This number will take on two values, first $m = k2^k$ returned as `number-of-bins` by procedure ESTIMATE-SIZE and then $m = 3 \cdot 2^k$ returned as `size` by the same procedure

ESTIMATE-SIZE, for $k > 3$. Because $m = k2^k$ in the first stage, and also $\texttt{size} = 3 \cdot 2^k > n$, by Lemma 19, we obtain that $m > \frac{n}{3} \lg \frac{n}{3}$ in the first stage, and that m is at least n in the following stages, with probability exponentially close to 1.

In the first stage, we throw $\ell_1 = n$ balls into at least $m = \frac{n}{3} \lg \frac{n}{3}$ bins, with large probability. Conditional on the event that there are at least these many bins, the probability that a given ball ends the stage as a singleton in a bin is

$$m \cdot \frac{1}{m}\Big(1 - \frac{1}{m}\Big)^{\ell_1 - 1} \geq 1 - \frac{\ell_1 - 1}{m} \geq 1 - \frac{n-1}{\frac{n}{3} \lg \frac{n}{3}} \geq 1 - \frac{4}{\lg n} \ ,$$

for sufficiently large n, where we used the Bernoulli's inequality. Let Y_1 be the number of singleton balls in the first stage. The expectancy of Y_1 satisfies

$$\mathbb{E}\left[Y_1\right] \geq \ell_1 \Big(1 - \frac{4}{\lg n}\Big) \ .$$

To estimate the deviation of Y_1 from its expected value $\mathbb{E}\left[Y_1\right]$ we use the bounded differences inequality [71, 75]. Let B_j be the bin of ball b_j, for $1 \leq j \leq \ell_1$. Then Y_1 is of the form $Y_1 = h(B_1, \ldots, B_{\ell_1})$, where h satisfies the Lipschitz condition with constant 2, because moving one ball to a different bin results in changing the value of h by at most 2 with respect to the original value. The bounded-differences inequality specialized to this instance is as follows, for any $d > 0$:

$$\Pr(Y_1 \leq \mathbb{E}\left[Y_1\right] - d\sqrt{\ell_1}) \leq \exp(-d^2/8) \ . \tag{7.8}$$

We employ $d = \lg n$, which makes the right-hand side of (7.8) asymptotically equal to $n^{-\Omega(\log n)}$. The number of balls ℓ_2 eligible for the second stage can be estimated as follows, this bound holding with probability $1 - n^{-\Omega(\log n)}$:

$$\ell_2 \leq \frac{4\ell_1}{\lg n} + \lg n \sqrt{\ell_1} = \frac{4\ell_1}{\lg n}\Big(1 + \frac{\lg^2 n}{4\sqrt{\ell_1}}\Big) \leq \frac{5n}{\lg n} \ , \tag{7.9}$$

for sufficiently large n.

In the second stage, we throw ℓ_2 balls into $m \geq n$ bins, with large probability. Conditional on the bound (7.9) holding, the probability that a given ball ends up as a singleton in

a bin is

$$m \cdot \frac{1}{m}\Big(1 - \frac{1}{m}\Big)^{\ell_2 - 1} \geq 1 - \frac{\ell_2 - 1}{m} \geq 1 - \frac{5}{\lg n} \ ,$$

where we used the Bernoulli's inequality. Let Y_2 be the number of singleton balls in the second stage. The expectancy of Y_2 satisfies

$$\mathbb{E}\left[Y_2\right] \geq \ell_2\Big(1 - \frac{5}{\lg n}\Big) \ .$$

To estimate the deviation of Y_2 from its expected value $\mathbb{E}\left[Y_2\right]$, we again use the bounded differences inequality, which specialized to this instance is as follows, for any $d > 0$:

$$\Pr(Y_2 \leq \mathbb{E}\left[Y_2\right] - d\sqrt{\ell_2}) \leq \exp(-d^2/8) \ . \tag{7.10}$$

We again employ $d = \lg n$, which makes the right-hand side of (7.10) asymptotically equal to $n^{-\Omega(\log n)}$. The number of balls ℓ_3 eligible for the third stage can be bounded from above as follows, which holds with probability $1 - n^{-\Omega(\log n)}$, :

$$\ell_3 \leq \frac{5\ell_2}{\lg n} + \lg n\sqrt{\ell_2} = \frac{5\ell_2}{\lg n}\Big(1 + \frac{\lg^2 n}{5\sqrt{\ell_2}}\Big) \leq \frac{6n}{\lg^2 n} \ , \tag{7.11}$$

for sufficiently large n.

Next, we generalize these estimates. In stages i, for $i \geq 2$, among the first $\mathcal{O}(\log n)$ ones, we throw balls into $m \geq n$ bins with large probability. Let ℓ_i be the number of balls eligible for such a stage i. We show by induction that ℓ_i, for $i \geq 3$, can be estimated as follows:

$$\ell_i \leq \frac{6n}{\lg^2 n} \cdot 2^{3-i} \tag{7.12}$$

with probability $1 - n^{-\Omega(\log n)}$. The estimate (7.11) provides the base of induction for $i = 3$. In the inductive step, we assume (7.12), and consider what happens during stage $i > 3$ in order to estimate the number of balls eligible for the next stage $i + 1$.

In stage i, we throw ℓ_i balls into $m \geq n$ bins, with large probability. Conditional on the bound (7.12), the probability that a given ball ends up single in a bin is

$$m \cdot \frac{1}{m}\Big(1 - \frac{1}{m}\Big)^{\ell_i - 1} \geq 1 - \frac{\ell_i - 1}{m} \geq 1 - \frac{6 \cdot 2^{3-i}}{\lg^2 n} \ ,$$

by the inductive assumption, where we also used the Bernoulli's inequality. If Y_i is the number of singleton balls in stage i, then its expectation $\mathbb{E}[Y_i]$ satisfies

$$\mathbb{E}[Y_i] \geq \ell_i\Big(1 - \frac{6 \cdot 2^{3-i}}{\lg^2 n}\Big) \ . \tag{7.13}$$

To estimate the deviation of Y_i from its expected value $\mathbb{E}[Y_i]$, we again use the bounded differences inequality, which specialized to this instance is as follows, for any $d > 0$:

$$\Pr(Y_i \leq \mathbb{E}[Y_i] - d\sqrt{\ell_i}) \leq \exp(-d^2/8) \ . \tag{7.14}$$

We employ $d = \lg n$, which makes the right-hand side of (7.14) asymptotically equal to $n^{-\Omega(\log n)}$. The number of balls ℓ_{i+1} eligible for the next stage $i+1$ can be estimated from above in the following way, the estimate holding with probability $1 - n^{-\Omega(\log n)}$:

$$\begin{aligned}
\ell_{i+1} &\leq \frac{6 \cdot 2^{3-i} \cdot \ell_i}{\lg^2 n} + \lg n \sqrt{\ell_i} \\
&= \frac{6 \cdot 2^{3-i} \cdot \ell_i}{\lg^2 n}\Big(1 + \frac{1}{6} 2^{i-3} \lg^3 n \cdot \ell_i^{-1/2}\Big) \\
&\leq \frac{6 \cdot 2^{3-i}}{\lg^2 n} \cdot \frac{6n}{\lg^2 n} \cdot 2^{3-i} \cdot \Big(1 + \frac{2^{(i-3)/2} \lg^4 n}{6\sqrt{6n}}\Big) \\
&\leq \frac{6n}{\lg^2 n} \cdot 2^{3-i} \cdot \Big(\frac{6 \cdot 2^{3-i}}{\lg^2 n} + \frac{2^{(3-i)/2} \lg^2 n}{\sqrt{6n}}\Big) \\
&\leq \frac{6n}{\lg^2 n} \cdot 2^{3-i} \cdot \Big(\frac{6}{\lg^2 n} + \frac{\lg^2 n}{\sqrt{6n}}\Big) \\
&\leq \frac{6n}{\lg^2 n} \cdot 2^{3-i-1} \ ,
\end{aligned}$$

for sufficiently large n that does not depend on i. For the event $Y_i \leq \mathbb{E}[Y_i] - d\sqrt{\ell_i}$ in the estimate (7.14) to be meaningful, it is sufficient if the following estimate holds:

$$\lg n \cdot \sqrt{\ell_i} = o(\mathbb{E}[Y_i]) \ .$$

This is the case as long as $\ell_i > \lg^3 n$, because $\mathbb{E}[Y_i] = \ell_i(1 + o(1))$ by (7.13).

To summarize at this point, as long as ℓ_i is sufficiently large, that is, $\ell_i > \lg^3 n$, the number of eligible balls decreases by at least a factor of 2 with probability that is at least $1 - n^{-\Omega(\log n)}$. It follows that the total number of eligible balls, summed over these stages, is $\mathcal{O}(n)$ with this probability.

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 7.5: 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 7.3 and 7.4, respectively. The variables `Last-Name` and `Collision-Detected` are shared.

After at most $\lg n$ such stages, the number of balls becomes at most $\lg^3 n$ with probability $1 - n^{-\Omega(\log n)}$. It remains to consider the stages when $\ell_i \leq \lg^3 n$, so that we throw at most $\lg^3 n$ balls into at least n bins. They all end up in singleton bins with a probability that is at least

$$\left(\frac{n - \lg^3 n}{n}\right)^{\lg^3 n} \geq \left(1 - \frac{\lg^3 n}{n}\right)^{\lg^3 n} \geq 1 - \frac{\lg^6 n}{n} \ ,$$

by the Bernoulli's inequality. So the probability of a collision is at most $\frac{\lg^6 n}{n}$. One stage without any collision terminates the process. If we repeat such stages $\lg n$ times, without even removing singleton balls, then the probability of collisions occurring in all these stages is at most

$$\left(\frac{\lg^6 n}{n}\right)^{\lg n} = n^{-\Omega(\log n)} \ .$$

The number of eligible balls summed over these final stages is only at most $\lg^7 n = o(n)$. □

The following Theorem summarizes the performance of algorithm COMMON-BOUNDED-MC (see the pseudocode in Figure 7.5) as a Monte Carlo one.

Theorem 10 *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 20. 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 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 19. The number of nonempty bins ever tested is at most dn, for some constant $d > 0$, by Lemma 20, with the suitably large probability. Applying the union bound results in 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 19. Each of the subsequent iterations takes time proportional to `size`, which is at most $6n$, again by Lemma 19. 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 20. 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 20.

Throwing one ball uses $\mathcal{O}(\log n)$ random bits, by Lemma 19. The number of throws is $\mathcal{O}(n)$ with the suitably large probability, by Lemma 20. □

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

$\mathcal{O}(n \log n)$, by Proposition 1, and the probability of error $n^{-\mathcal{O}(1)}$, by Proposition 3.

7.4 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 7.3, 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.

Algorithm COMMON-UNBOUNDED-MC has its pseudocode given in Figure 7.7. The algorithm is structured as a repeat loop. An iteration starts by invoking procedure GAUGE-SIZE, whose pseudocode is in Figure 7.6. 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 follows by assigning to 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 7.6. 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 func-

Procedure GAUGE-SIZE-MC

$k \leftarrow 1$

`repeat`

$k \leftarrow r(k)$

$\texttt{bin}_v \leftarrow$ random integer in $[1, 2^k]$

`until` the number of selected values of variable `bin` is $\leq 2^k/\beta$

`return` ($\lceil 2^{k+1}/\beta \rceil$)

Figure 7.6: 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 7.7, and an increasing function $r(k)$ is also a parameter.

tion $r(k) = k + 1$ and the other is function $r(k) = 2k$.

Lemma 21 *If $r(k) = k+1$ then the value of* `size` *as returned by* GAUGE-SIZE-MC *satisfies* `size` $\leq 2n$ *with certainty and the inequality* `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* `size` $\leq 2\beta n^2$ *with certainty and* `size` $\geq \beta n^2/2$ *with probability* $1 - \beta^{-n/3}$.

Proof: We model procedure's execution by an experiment of throwing n balls into 2^k bins. If the parameter function $r(k)$ is $r(k) = k + 1$ then we consider all possible consecutive values of k starting from $k = 2$, such that $k = i + 1$ in the ith iteration of the repeat-loop. If parameter $r(k)$ is function $r(k) = 2k$ then k takes on only the powers of 2.

There are at most n bins occupied in any such an experiment. Therefore, the procedure returns by the time the inequality $2^k/\beta \geq n$ holds and k is considered as determining the range of bins. It follows that if $r(k) = k + 1$ then the returned value $\lceil 2^{k+1}/\beta \rceil$ is at most $2n$. If $r(k) = 2k$ then the worst error in estimating occurs when $2^i/\beta = n - 1$ for some i that is a power of 2. Then the returned value is $2^{2i}/\beta = (\beta(n-1))^2/\beta$, which is at most $2\beta n^2$, this occurring with probability $1 - \beta^{-n/3}$.

Given 2^k bins, we estimate the probability that the number of occupied bins is at most

$2^k/\beta$. It is

$$\binom{2^k}{2^k/\beta}\Big(\frac{2^k/\beta}{2^k}\Big)^n \le \Big(\frac{2^k e}{2^k/\beta}\Big)^{2^k/\beta} \cdot \frac{1}{\beta^n} = (e\beta)^{2^k/\beta} \cdot \beta^{-n} \ .$$

Next, we identify a range of values of k for which this probability is exponentially close to 0 with respect to n.

To this end, let $0 < \rho < 1$ and let us consider the inequality

$$(e\beta)^{2^k/\beta} \cdot \beta^{-n} < \rho^n \ . \tag{7.15}$$

It is equivalent to the following one

$$\frac{2^k}{\beta}(1 + \ln\beta) - n\ln\beta < n\ln\rho \ ,$$

by taking logarithms of both sides. This in turn is equivalent to

$$\frac{2^k}{\beta}(1 + \ln\beta) < n\Big(\ln\beta - \ln\frac{1}{\rho}\Big) \ . \tag{7.16}$$

Let us choose $\rho = \beta^{-1/2}$ in (7.16). Then (7.15) specialized to this particular ρ is equivalent to the following inequality $\frac{2^k}{\beta}(1 + \ln\beta) < n\frac{\ln\beta}{2}$. This in turn leads to the estimate

$$2^k < n \cdot \frac{\ln\beta}{2} \cdot \frac{\beta}{1 + \ln\beta} < \frac{\beta}{2} \cdot n \ ,$$

which means $2^{k+1}/\beta < n$. When k satisfies this inequality then the probability of returning is at most $\beta^{-n/2}$. There are $\mathcal{O}(\log n)$ such values of k considered by the procedure, so it returns for one of them with probability at most

$$\mathcal{O}(\log n) \cdot \beta^{-n/2} < \beta^{-n/3} \ ,$$

for sufficiently large n.

Therefore, with probability at least $1 - \beta^{-n/3}$, the returned value $\lceil 2^{k+1}/\beta \rceil$ is at least as large as determined the first considered k that satisfies $2^{k+1}/\beta \ge n$. If $r(k) = k + 1$ then all the possible exponents k are considered, so the returned value $\lceil 2^{k+1}/\beta \rceil$ is at least n with probability $1-\beta^{-n/3}$. If $r(k) = 2k$ then the worst error of estimating n occurs when $2^{i+1}/\beta = n-1$ for some i that is a power of 2. Then the returned value is $2^{2i+1}/\beta = 2\cdot(\beta(n-1)/2)^2/\beta$, which is is at least $\beta n^2/2$, this occurring with probability $1 - \beta^{-n/3}$. □

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

We discuss performance of algorithm COMMON-UNBOUNDED-MC (see the pseudocode in Figure 7.7) by referring to analysis of a related algorithm COMMON-UNBOUNDED-LV given in Section 6.4. We consider a *β-process with verifications*, which is defined as follows. The process proceeds through stages. The first stage starts with placing n balls into $3\,\texttt{size}$ bins. For any of subsequent stages, for each multiple bins and for each ball in such a bin we perform a Bernoulli trial with the probability $\frac{1}{2}$ of success, which represents the outcome of procedure VERIFY-COLLISION. A success in a trial is referred to as a *positive verification* otherwise it is a *negative* one. If at least one positive verification occurs for a ball in a multiple bin then all the balls in this bin are relocated in this stage to bins selected uniformly at random and independently for each such a ball, otherwise the balls stay put in this bin until the next stage. The process terminates when all balls are singleton.

Lemma 22 *For any number $a > 0$ there exists $\beta > 0$ such that the β-process with verifica-*

tions terminates within $\beta \lg n$ stages with all of them comprising the total of $\mathcal{O}(n)$ ball throws with probability at least $1 - n^{-a}$.

Proof: We use the respective Lemma 11 in Section 6.4. The constant 3 determining our β-process with verifications corresponds to $1 + \beta$ in Section 6.4. The corresponding β-process in verifications considered in Section 6.4 is defined by referring to known n. We use the approximation `size` instead, which is at least as large as n with probability $1 - \beta^{-n/3}$, by Lemma 21 just proved. By Section 6.4, our β-process with verifications does not terminate within $\beta \lg n$ stages when `size` $\geq n$ with probability at most n^{-2a} and the inequality `size` $\geq n$ does not hold with probability at most $\beta^{-n/3}$. Therefore the conclusion we want to prove does not hold with probability at most $n^{-2a} + \beta^{-n/3}$, which is at most n^{-2a} for sufficiently large n. □

The following Theorem summarizes the performance of algorithm COMMON-UNBOUNDED-MC (see the pseudocode in Figure 7.7) 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 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 $1 - n^{-a}$. 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 $\mathcal{O}(\log n)$.*

Proof: For a given $a > 0$, let us take β that exists by Lemma 22. When the β-process with verifications terminates then this models assigning unique names by the algorithm. It follows that one iteration of the repeat-loop results in algorithm terminating with proper names assigned with probability $1 - n^{-a}$. One iteration of the main repeat-loop does not result in

termination with probability at most n^{-a}, so i iterations are not sufficient to terminate with probability at most n^{-ia}. This converges to 0 with increasing i so the algorithm terminates almost surely.

The performance metrics rely mostly on Lemma 21. We consider two cases, depending on which function $r(k)$ is used.

If $r(k) = k + 1$ then procedure Gauge-Size-MC considers all the consecutive values of k up to $\lg n$, and for each such k, throwing a ball requires k random bits. We obtain that procedure Gauge-Size-MC uses $\mathcal{O}(n \log^2 n)$ random bits. Similarly, to compute the number of selected values in an iteration of the main repeat-loop of this procedure takes time $\mathcal{O}(k)$, for the corresponding k, so this procedure takes $\mathcal{O}(\log^2 n)$ time. The value of `size` satisfies `size` $\leq 2n$ with certainty. Therefore, $\mathcal{O}(n)$ memory registers are ever needed and one throw of a ball uses $\mathcal{O}(\log n)$ random bits, after `size` has been computed. It follows that one iteration of the main repeat-loop of the algorithm, after procedure Gauge-Size-MC has been completed, uses $\mathcal{O}(n \log n)$ random bits, by Lemmas 21 and 22, and takes $\mathcal{O}(\log n)$ time. Since one iteration of the main repeat-loop suffices with probability $1 - n^{-a}$, the overall time is dominated by the time performance of procedure Gauge-Size-MC.

If $r(k) = 2k$ then procedure Gauge-Size-MC considers all the consecutive powers of 2 as values of k up to $\lg n$, and for each such k, throwing a ball requires k random bits. Since the values k form a geometric progression, procedure Gauge-Size-MC uses $\mathcal{O}(\log n)$ random bits per processor. Similarly, to compute the number of selected values in an iteration of the main repeat-loop of this procedure takes time $\mathcal{O}(k)$, for the corresponding k that increase geometrically, so this procedure takes $\mathcal{O}(\log n)$ time. The value of `size` satisfies `size` $\leq 2n$ with certainty. By Lemma 21, $\mathcal{O}(n^2)$ memory registers are ever needed, so one throw of a ball uses $\mathcal{O}(\log n)$ random bits. One iteration of the main repeat-loop, after procedure Gauge-Size-MC has been completed, uses $\mathcal{O}(n \log n)$ random bits, by Lemmas 21 and 22, and takes $\mathcal{O}(\log n)$ time. □

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 2, 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 1. In the other case of $r(k) = 2k$, the expected time $\mathcal{O}(\log n)$ is optimal, by Theorem 3, the expected number of random bits $\mathcal{O}(n \log n)$ is optimal, by Proposition 1, 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.

7.5 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 register are provably optimal with respect to the following three performance metrics: expected time, expected number of generated random bits and probability of error.

8. Naming a Channel with Beeps

In this section, we consider anonymous channel with beeping. We present names can be assigned to the anonymous stations by a Las Vegas and a Monte Carlo naming algorithms.

8.1 A Las Vegas Algorithm

We give a Las Vegas naming algorithm for the case when n is known. The idea is to have stations choose rounds to beep from a segment of integers. As a convenient probabilistic interpretation, these integers are interpreted as bins, and after selecting a bin a ball is placed in the bin. The algorithm proceeds by considering all the consecutive bins. First, a bin is verified to be nonempty by making the owners of the balls in the bin beep. When no beep is heard then the next bin is considered, otherwise the nonempty bin is verified for collisions. Such a verification is performed by $\mathcal{O}(\log n)$ consecutive calls of procedure Detect-Collision. When a collision is not detected then the stations that placed their balls in this bin assign themselves the next available name, otherwise the stations whose balls are in this bin place their balls in a new set of bins. When each station has a name assigned, we verify if the maximum assigned name is n. If this is the case then the algorithm terminates, otherwise we repeat. The algorithm is called Beep-Naming-LV, its pseudocode is in Figure 8.1.

Algorithm Beep-Naming-LV is analyzed by modeling its executions by a process of throwing balls into bins, which we call the *ball process*. The process proceeds through stages. There are n balls in the first stage. When a stage begins and there are some i balls eligible for the stage then the number of used bins is $i \lg n$. Each ball is thrown into a randomly selected bin. Next, balls that are singleton in their bins are removed and the remaining balls that participated in collisions advance to the next stage. The process terminates when no eligible balls remain.

Lemma 23 *The number of times a ball is thrown into a bin during an execution of the ball process that starts with n balls is at most $3n$ with probability at least $1 - e^{-n/4}$.*

Proof: In each stage, we throw some k balls into at least $k \lg n$ bins. The probability that

Algorithm BEEP-NAMING-LV

```
repeat
    counter ← 0 ; left ← 1 ; right ← n lg n ; name_v ← null
    repeat
        slot_v ← random number in the interval [left, right]
        for i ← left to right do
            if i = slot_v then
                beep
                if a beep was just heard then
                  collision ← false
                  for j ← 1 to β lg n do
                      if DETECT-COLLISION then collision ← true
                  if not collision then
                      counter ← counter + 1
                      name_v ← counter
        if name_v = null then beep
        if a beep was just heard then
            left ← counter
            right ← (n − counter) lg n
    until no beep was heard in the previous round
until counter = n
```

Figure 8.1: A pseudocode for a station v. The number of stations n is known. Constant $\beta > 1$ is a parameter determined in the analysis. Procedure DETECT-COLLISION has its pseudocode in Figure 4.2. The variable `name` is to store the assigned identifier.

a given ball ends up singleton in a bin is at least

$$1 - \frac{k}{k \lg n} = 1 - \frac{1}{\lg n} \ ,$$

which we denote as p. A ball is thrown repeatedly in consecutive iterations until it lands single in a bin. Our immediate concern is the number of trials to have all balls as singletons in their bins.

Suppose that we perform some m independent Bernoulli trials, each with probability p of success, and let X be the number of successes. We show next that $m = \Theta(n)$ suffices with large probability to have the inequality $X \geq n$.

The expected number of successes is $\mathbb{E}[X] = \mu = pm$. We use the Chernoff bound in the form

$$\Pr(X < (1-\varepsilon)\mu) < e^{-\varepsilon^2\mu/2} , \tag{8.1}$$

for any $0 < \varepsilon < 1$; see [77]. It suffices to have the inequality $(1-\varepsilon)\mu \geq n$. Let us set $\varepsilon = \frac{1}{2}$, so that it suffices to have $\mu \geq 2n$ or $pm \geq 2n$, which is extended into the following form:

$$\left(1 - \frac{1}{\lg n}\right) \cdot m \geq 2n . \tag{8.2}$$

If we choose $m = 3n$ then inequality (8.2) holds for sufficiently large n.

The probability that this inequality (8.2) does not hold is estimated from above by (8.1). Here we have that $\mu \geq 2n$ so the right-hand side of (8.1) is $e^{-n/4}$. □

We proceed to Theorem 12, which summarizes all the good properties of algorithm Beep-Naming-LV, in particular that the algorithm is Las Vegas. In the proof, we model executions of the algorithm as that of the ball process starting with n balls. The main difference between the ball process and the algorithm is that collisions of balls in bins are detected with certainty, by the specification of the process, while in the algorithm collisions between tentative names might be overlooked with some positive probability.

Theorem 12 *Algorithm* Beep-Naming-LV*, for any $\beta > 0$, terminates almost surely and there is no error when it terminates. For each $a > 0$, there exists $\beta > 1$ and $c > 0$ such that the algorithm assigns unique names, works in time at most $cn \lg n$, and uses at most $cn \lg n$ random bits, all these properties holding with probability at least $1 - n^{-a}$, for sufficiently large n.*

Proof: Consider an iteration of the main repeat-loop. An error can occur in this iteration only when there is a collision that is not detected by procedure Detect-Collision in none of its $\beta \lg n$ calls. Such an error results in duplicate names, so that the number of assigned different names is smaller than n. The maximum name assigned in an iteration is the value of the variable `counter`, which has the same value at each station. The algorithm terminates

by having an iteration that produces `counter` $= n$, but then there are no repetitions among the names, and so there is no error.

Next we show that termination is a sure event. Consider an iteration of the main repeat-loop. There are n balls and each of them is kept thrown until either it is not involved in a collision or there is a collision but it is not detected. Eventually each ball is left to reside in its bin with probability 1. This means that each iteration ends almost surely.

We introduce the notation for two events in an iteration of the main repeat-loop. Let A be the event that there is a collision that passes undetected. The iteration fails to assign proper names if and only if event A holds. Let B be the event that the total number of throwing balls into bins is at most $3n$. We denote by $\neg E$ the complements of an event E. We have that $\Pr(\neg B) \leq e^{-n/4}$, by Lemma 23.

When a ball lands in a bin then it is verified for a collision $\beta \lg n$ times. If there is a collision then it passes undetected with probability at most $n^{-\beta}$. This is because one call of procedure DETECT-COLLISION detects a collision with probability at least $\frac{1}{2}$, by Lemma 2, in which $m = 1$ and $k \geq 2$.

We estimate the probability of the event that an iteration fails to assign proper names, which is the same as of event A. This is accomplished as follows:

$$\begin{aligned}
\Pr(A) &= \Pr(A \cap B) + \Pr(A \cap \neg B) \\
&= \Pr(A \mid B) \cdot \Pr(B) + \Pr(A \mid \neg B) \cdot \Pr(\neg B) \\
&\leq \Pr(A \mid B) + \Pr(\neg B) \\
&\leq 3n \cdot n^{-\beta} + e^{-n/4} \ , \qquad (8.3)
\end{aligned}$$

where we used the union bound to obtain the last line (8.3). It follows that at least i iterations are needed with probability at most $(e^{-n/4} + 3n^{1-\beta})^i$, which converges to 0 as i grows unbounded, assuming only that $\beta > 1$ and n is sufficiently large.

Let us consider the event $\neg A \cap B$, which occurs when balls are thrown at most $3n$ times and all collisions are detected, when modeling an iteration of the main repeat loop. The

probability that event $\neg A \cap B$ holds can be estimated from below as follows:

$$\begin{aligned}\Pr(\neg A \cap B) &= \Pr(\neg A \mid B) \cdot \Pr(B) \\ &\geq (1 - 3n^{1-\beta}) \cdot (1 - e^{-n/4}) \\ &\geq 1 - 3n^{1-\beta} - e^{-n/4}(1 - 3n^{1-\beta}) \ . \end{aligned} \tag{8.4}$$

This bound (8.4) is at least $1 - n^{-a}$ for sufficiently large $\beta > 1$, when also n is large enough.

Bound (8.4) holds for the first iteration of the main repeat loop. So with probability at least $1 - n^{-a}$ the first iteration assigns proper names with at most $3n$ balls thrown in total. Let us assume that this event occurs. Then the whole execution takes time at most $cn \lg n$, for a suitably large $c > 0$. This is because procedure DETECT-COLLISION is executed at most $3\beta n \lg n$ times, and each of its calls takes two rounds. One assignment of a value to variable `slot` requires $\lg(n \lg n) < 2 \lg n$ bits, for sufficiently large n. There are at most $3n$ such assignments, for a total of at most $cn \lg n$ random bits, for a suitably large $c > 0$. □

Algorithm BEEP-NAMING-LV runs in the optimal expected time $\mathcal{O}(n \log n)$, by Proposition 6, and it uses the optimum expected number of random bits $\mathcal{O}(n \log n)$, by Proposition 5, these propositions given in Section 5.3.

8.2 A Monte Carlo Algorithm

We give a randomized naming algorithm for the case when n is unknown. In view of Proposition 8, no Las Vegas algorithm exists in this case, so we develop a Monte Carlo one.

The algorithm again can be interpreted as repeatedly throwing balls into bins and verifying for collisions. A bin is determined by a string of some k bits. Each station chooses one such a string randomly. The algorithm proceeds to repeatedly identify the smallest lexicographically string among those not considered yet. This is accomplished by procedure NEXT-STING which operates as a search implemented by using beeps. Having identified a nonempty bin, all the stations that placed their balls into this bin verify if there is a collision in this bin by calling DETECT-COLLISION a suitably large number of times. In case no collision has been detected, the stations whose balls are in the bin assign themselves the

consecutive available name as a temporary one. This continues until all the balls have been considered. If no collision has ever been detected in the current stage, then the algorithm terminates and the temporary names are considered as the final assigned names, otherwise the algorithm proceeds to the next stage.

Next, we specify procedure NEXT-STRING. It operates as a radix search to identify the smallest string of bits by considering consecutive bit positions. It uses two variables `my-string` and k, where k is the length of the bit strings considered and $\texttt{my-string}_v$ is the string of k bits generated by station v. The procedure begins by setting to 1 all bit positions in variable `string`, which has k such bit positions. Then the consecutive bit positions $i = 1, 2, \ldots, k$ are considered one by one. For a given bit position i, all the stations v, that still can possibly have the smallest string and whose bit on position i in $\texttt{my-string}_v$ is 0, do beep. This determines the first i bits of the smallest string, because if a beep is heard then the ith bit of the smallest string is 0 and otherwise it is 1. This is recorded by setting the ith bit position in the variable `string` to the determined bit. The stations eligible for beeping, if their ith bit is 0, are those whose strings agree on the first $i-1$ positions with the smallest string. After all k bit positions have been considered, the variable `string` is returned.

Procedure NEXT-STRING has its pseudocode in Figure 8.2. Its relevant property is summarized as the following lemma.

Lemma 24 *Procedure* NEXT-STRING *returns the smallest lexicographically string among the non-null string values of the private copies of the variable* ***my-string****.*

Proof: The string that is output is obtained by processing all the input strings `my-string` through consecutive bit positions. We show the invariant that after i bits have been considered, for $0 \leq i \leq k$, then the bits on these positions make the prefix of the first i bits of the smallest string.

The invariant is shown by induction on i. When $i = 1$ then the bits on previously considered positions make an empty string, as no positions have been considered yet, and the empty string is a prefix of the smallest string. Suppose that the invariant holds for all i

Procedure NEXT-STRING

```
string ← a string of k bit positions, with all of them set to 1
for i ← 1 to k do
    if (my-string_v matches string on the first i − 1 bit positions)
                    and (the ith bit of my-string_v is 0)
        then beep
    if a beep was heard in the previous round then
        set the ith bit of string to 0
return (string)
```

Figure 8.2: A pseudocode for a station v. This procedure is used by algorithm BEEP-NAMING-MC. The variables `my-string` and k are the same as those in the pseudocode in Figure 8.3.

such that $0 \leq i < k$, and consider the stations whose variable `my-string` has the same bits on these first i positions as variable `string`. This set includes the station v with the smallest `my-string` by the inductive assumption. If the bit on the $(i+1)$st position of $\texttt{my-string}_v$ is 0 then v beeps and `string` has its bit on position $i+1$ set to 0. Otherwise there is no station with 0 on the $(i+1)$st position of $\texttt{my-string}_v$, because $\texttt{my-string}_v$ is smallest. Then there is no beep and 1 at position $i+1$ in `string` is not modified. This completes the proof of the invariant.

The procedure NEXT-STRING terminates after k bit positions have been processed. The proved invariant for $i = k$ means that the smallest $\texttt{my-string}_v$ and the final value of the variable `string` are identical. □

The naming algorithm we develop is called BEEP-NAMING-MC. Its pseudocode is in Figure 8.3. The algorithm proceeds through stages, where a stage is implemented by an iteration of the main repeat-loop in the pseudocode. The number of bins in the ith stage is 2^k where $k = 2^i$. The variable k is doubled in the beginning of each iteration of the main loop. During a stage, first the next bin with a ball is identified by calling procedure NEXT-

Algorithm BEEP-NAMING-MC

```
k ← 1
repeat
    k ← 2k
    collision ← false
    counter ← 0
    my-string_v ← a random string of k bits
    repeat
        if my-string_v = smallest-string then
        if my-string_v ≠ null then smallest-string ← NEXT-STRING
            for i ← 1 to β · k do
                if DETECT-COLLISION then collision ← true
            if not collision then
                counter ← counter + 1
                name_v ← counter
            my-string_v ← null
        if my-string_v ≠ null then beep
    until no beep was heard in the previous round
until not collision
```

Figure 8.3: A pseudocode for a station v. Constant $\beta > 0$ is an integer parameter determined in the analysis. Procedure DETECT-COLLISION has its pseudocode in Figure 4.2 and procedure NEXT-STRING has its pseudocode in Figure 8.2. The variable `name` is to store the assigned identifier.

STRING. Next, this bin is verified for collisions by calling procedure DETECT-COLLISION βk times, for a constant $\beta > 0$, which is a parameter to be settled in analysis. During such a verification, the stations whose balls are in this bin participate only.

The next theorem summarizes the good properties of algorithm BEEP-NAMING-MC. In particular, that it is a Monte Carlo algorithm with a suitably small probability of error.

Theorem 13 *Algorithm* BEEP-NAMING-MC*, for any $\beta > 0$, 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 $cn \lg n$, and uses at most $cn \lg n$ random bits, all these properties holding with probability at least $1 - n^{-a}$.*

Proof: We interpret an iteration of the outer repeat-loop as a stage in a process of throwing

n balls into 2^k bins and verifying βk times for collisions. The string selected by a station is the name of the bin. When at least one collision is detected then k gets incremented and another iteration is performed. An error occurs when there is a collision but it is not detected.

Next we estimate from above the probability of not detecting a collision. To this end, we consider two cases, depending on which of the inequalities $2^k < n$ or $2^k \geq n$ hold for a given k.

In the first case, when $2^k < n$, collisions occur with certainty, by the pigeonhole principle. Let m be the number of occupied bins. This results in $m \leq 2^k$ verifications performed, one for each bin, where procedure DETECT-COLLISION is called βk times per verification. By Lemma 2, the probability of not detecting a collision, with just one call of DETECT-COLLISION occurring in each of these verifications, is at most

$$2^{-n+m} \leq 2^{-n+2^k} .$$

When βk calls of DETECT-COLLISION occur in each verification, as is the case by the design of the algorithm, the probability of not detecting a collision is at most

$$2^{(-n+2^k)\beta k} .$$

Intuitively at this point, since $2^k < n$, this probability is maximized for $n = 2^k + 1$ and it is about $2^{-\beta k} \approx n^{-\beta}$, as $k \approx \lg n$.

A precise argument to obtain an estimate is by considering two sub-cases, and is as follows. If it is the sub-case that $2^k < n/2$, then

$$2^{(-n+2^k)\beta k} = 2^{-\Omega(n)} .$$

Otherwise, when it is the sub-case that $n > 2^k \geq n/2$, then $n - 2^k \geq 1$ and $k \geq \lg n - 1$, so that we obtain the following estimate:

$$2^{(-n+2^k)\beta k} \geq 2^{-\beta(\lg n - 1)} \geq 2^{\beta} n^{-\beta} .$$

We obtained the following two estimates: $2^{-\Omega(n)}$ and $2^{\beta}n^{-\beta}$, of which the latter is larger, for sufficiently large n. It is sufficient to take $\beta > a$, as then the inequality $2^{\beta}n^{-\beta} < n^{-a}$ holds for sufficiently large n.

The second case occurs when $2^k \geq n$. This implies that $k \geq \lg n$. When a collision occurs in a bin, then it is verified by at least $\beta \lg n$ calls of procedure DETECT-COLLISION. This gives probability at most $n^{-\beta}$ of not detecting one such a collision. Multiple bins with collisions make the probability of not detecting any of them even smaller. Now it is enough to take $\beta > a$, as then $n^{-\beta} < n^{-a}$ holds.

This completes estimating the probability of error by at most n^{-a}, for sufficiently large β, and all correspondingly large n.

Next, we estimate the probability that the running time is $\mathcal{O}(n \log n)$. Let us consider a stage with sufficiently many bins, say, when $k = d \lg n$ for $d > 2$. Then the number of bins is $2^k = n^d$. The probability that there is no collision at all in this stage is at least

$$\left(1 - \frac{n}{n^d}\right)^n \geq 1 - \frac{n}{n^{d-1}} = 1 - n^{-d+2} \ . \tag{8.5}$$

Choosing $d = a + 2$ we obtain that the algorithm terminates by the iteration of the outer repeat-loop when $k = d \lg n$ with probability at least $1 - n^{-a}$. One iteration of the outer repeat loop, for some k, is proportional to $k \cdot n$. The total time spent up to and including $k = d \lg n$ is proportional to

$$\sum_{i=1}^{\lg((a+2)\lg n)} 2^i \cdot n \leq n \cdot 2(a+2) \lg n = \mathcal{O}(n \log n) \tag{8.6}$$

with probability at least $1 - n^{-a}$.

The number of bits generated up to and including the iteration for $k = d \lg n$ is also proportional to (8.6). This is because the number of bits generated in one iteration of the main repeat-loop is proportional to $k \cdot n$, similarly as the running time.

To show that the algorithm terminates almost surely, it is sufficient to demonstrate that the probability of a collision converges to zero with k increasing. The probability of no

collision for $k = d \lg n$ is at most n^{-d+2}, by (8.6). If k grows to infinity then $d = k/\lg n$ increases to infinity as well, and then n^{-d+2} converges to 0 as a function of d. □

Algorithm Beep-Naming-MC is optimal with respect to the following performance measures: the expected running time $\mathcal{O}(n \log n)$, by Proposition 6, the expected number of used random bits $\mathcal{O}(n \log n)$, by Proposition 5, and the probability of error, as determined by the number of used bits, by Proposition 7.

8.3 Conclusion

We considered a channel in which a synchronized beeping is the only means of communication. We showed that names can be assigned to the anonymous stations by randomized algorithms. The algorithms are either Las Vegas or Monte Carlo, depending on whether the number of stations n is known or not, respectively. The performance characteristics of the two algorithms, such as the running time, the number of random bits, and the probability of error, are proved to be optimal.

9. Open Problems and Future Work

Here we give some of the open problems and future work. The algorithms cover the "boundary" cases for the anonymous synchronous PRAM. One case is about a minimum amount of shared memory, that is, when only a constant number of shared memory cells are available. The other case is about a minimum expected running time, that is, when the expected running time is $\mathcal{O}(\log n)$; such performance requires a number of shared registers that grows unbounded with n. It would be interesting to have these results generalized by investigating naming on a PRAM when the number of processors and the number of shared registers are independent parameters of the model.

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: expected time, expected number of generated random bits and probability of error.

The algorithms we developed for beeping channels rely in an essential manner on synchronization of the channel. It would be interesting to consider an anonymous asynchronous beeping channel and investigate how to assign names to stations in such a communication environment.

REFERENCES

[1] Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. *Distributed Computing*, 26(4):195–208, 2013.

[2] Yehuda Afek, Noga Alon, Omer Barad, Eran Hornstein, Naama Barkai, and Ziv Bar-Joseph. A biological solution to a fundamental distributed computing problem. *Science*, 331(6014):183–185, 2011.

[3] Yehuda Afek and Yossi Matias. Elections in anonymous networks. *Information and Computation*, 113(2):312–330, 1994.

[4] 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.

[5] Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the 24th International Symposium on Distributed Computing (DISC)*, volume 6343 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2010.

[6] Dana Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th ACM Symposium on Theory of Computing (STOC)*, pages 82–93, 1980.

[7] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.

[8] Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. On the power of anonymous one-way communication. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005)*, volume 3974 of *Lecture Notes in Computer Science*, pages 396–411. Springer, 2006.

[9] Dana Angluin, James Aspnes, Michael J. Fischer, and Hong Jiang. Self-stabilizing population protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 3(4), 2008.

[10] 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.

[11] James Aspnes and Eric Ruppert. An introduction to population protocols. *Bulletin of the EATCS*, 93:98–117, 2007.

[12] 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.

[13] 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.

[14] Hagit Attiya and Faith Ellen. *Impossibility Results for Distributed Computing*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2014.

[15] Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2):162–183, 2002.

[16] Hagit Attiya and Marc Snir. Better computing on the anonymous ring. *Journal of Algorithms*, 12(2):204–238, 1991.

[17] Hagit Attiya, Marc Snir, and Manfred K. Warmuth. Computing on an anonymous ring. *Journal of the ACM*, 35(4):845–875, 1988.

[18] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley, 2nd edition, 2004.

[19] Paul Beame. Limits on the power of concurrent-write parallel machines. *Information and Computation*, 76(1):13–28, 1988.

[20] 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.

[21] 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.

[22] Philipp Brandes, Marcin Kardas, Marek Klonowski, Dominik Pająk, and Roger Wattenhofer. Approximating the size of a radio network in beeping model. In *Proceedings of the* 23*rd International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, 2016. To appear.

[23] 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.

[24] Jérémie Chalopin, Yves Métivier, and Thomas Morsellino. Enumeration and leader election in partially anonymous and multi-hop broadcast networks. *Fundamenta Informatica*, 120(1):1–27, 2012.

[25] Bogdan S. Chlebus. Randomized communication in radio networks. In Panos M. Pardalos, Sanguthevar Rajasekaran, John H. Reif, and Jose D. P. Rolim, editors, *Handbook of Randomized Computing*, volume I, pages 401–456. Kluwer Academic Publishers, 2001.

[26] Bogdan S. Chlebus, Gianluca De Marco, and Muhammed Talo. Anonymous processors with synchronous shared memory: Las Vegas algorithms. Submitted to a journal.

[27] Bogdan S. Chlebus, Gianluca De Marco, and Muhammed Talo. Anonymous processors with synchronous shared memory: Monte Carlo algorithms. Submitted to a journal.

[28] Bogdan S. Chlebus, Gianluca De Marco, and Muhammed Talo. Naming a channel with beeps. Submitted to a journal.

[29] Bogdan S. Chlebus, Krzysztof Diks, and Andrzej Pelc. Waking up an anonymous faulty network from a single source. In *Proceedings of the* 27*th Hawaii International Conference on System Sciences (HICSS)*, pages 187–193. IEEE, 1994.

[30] 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.

[31] Stephen A. Cook, Cynthia Dwork, and Rüdiger Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal of Computing*, 15(1):87–97, 1986.

[32] Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In *Proceedings of the* 24*th International Symposium on Distributed Computing (DISC)*, volume 6343 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2010.

[33] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory.* Wiley, 2nd edition, 2006.

[34] Artur Czumaj and Peter Davies. Communicating with beeps. *CoRR*, abs/1505.06107, 2015.

[35] Julius Degesys, Ian Rose, Ankit Patel, and Radhika Nagpal. DESYNC: self-organizing desynchronization and TDMA on wireless sensor networks. In *Proceedings of the* 6*th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 11–20. ACM, 2007.

[36] Dariusz Dereniowski and Andrzej Pelc. Leader election for anonymous asynchronous agents in arbitrary networks. *Distributed Computing*, 27(1):21–38, 2014.

[37] Yoann Dieudonné and Andrzej Pelc. Anonymous meeting in networks. *Algorithmica*, 74(2):908–946, 2016.

[38] Krzysztof Diks, Evangelos Kranakis, Adam Malinowski, and Andrzej Pelc. Anonymous wireless rings. *Theoretical Computer Science*, 145(1&2):95–109, 1995.

[39] Ömer Eğecioğlu and Ambuj K. Singh. Naming symmetric processes using shared variables. *Distributed Computing*, 8(1):19–38, 1994.

[40] Yuval Emek, Jochen Seidel, and Roger Wattenhofer. Computability in anonymous networks: Revocable vs. irrecovable outputs. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP), Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 183–195. Springer, 2014.

[41] Yuval Emek and Roger Wattenhofer. Stone age distributed computing. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 137–146, 2013.

[42] William Feller. *An Introduction to Probability Theory and Its Applications*, volume I. Wiley, 3rd edition, 1968.

[43] Faith E. Fich, Friedhelm Meyer auf der Heide, Prabhakar Ragde, and Avi Wigderson. One, two, three... infinity: Lower bounds for parallel computation. In *Proceedings of the 17th ACM Symposium on Theory of Computing (STOC)*, pages 48–58, 1985.

[44] Faith E. Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, 2003.

[45] Paola Flocchini, Evangelos Kranakis, Danny Krizanc, Flaminia L. Luccio, and Nicola Santoro. Sorting and election in anonymous asynchronous rings. *Journal of Parallel and Distributed Computing*, 64(2):254–265, 2004.

[46] Roland Flury and Roger Wattenhofer. Slotted programming for sensor networks. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 24–34. ACM, 2010.

[47] Klaus-Tycho Förster, Jochen Seidel, and Roger Wattenhofer. Deterministic leader election in multi-hop beeping networks. In *Proceedings of the 28th International Symposium on Distributed Computing (DISC)*, volume 8784 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 2014.

[48] Pierre Fraigniaud, Andrzej Pelc, David Peleg, and Stephane Perennes. Assigning labels in an unknown anonymous network with a leader. *Distributed Computing*, 14(3):163–183, 2001.

[49] Mohsen Ghaffari and Bernhard Haeupler. Near optimal leader election in multi-hop radio networks. In *Proceedings of the 24th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 748–766. SIAM, 2013.

[50] Mohsen Ghaffari, Bernhard Haeupler, and Majid Khabbazian. Randomized broadcast in radio networks with collision detection. *Distributed Computing*, 28(6):407–422, 2015.

[51] Seth Gilbert and Calvin C. Newport. The computational power of beeps. In *Proceedings of the 29th International Symposium on Distributed Computing (DISC)*, volume 9363 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2015.

[52] Leszek Gąsieniec, Evangelos Kranakis, Danny Krizanc, and X. Zhang. Optimal memory rendezvous of anonymous mobile agents in a unidirectional ring. In *Proceedings of the 32nd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 3831 of *Lecture Notes in Computer Science*, pages 282–292. Springer, 2006.

[53] Christian Glacet, Avery Miller, and Andrzej Pelc. Time vs. information tradeoffs for leader election in anonymous trees. In *Proceedings of the 27th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 600–609. SIAM, 2016.

[54] Olga Goussevskaia, Yvonne Anne Pignolet, and Roger Wattenhofer. Efficiency of wireless networks: Approximation algorithms for the physical interference model. *Foundations and Trends in Networking*, 4(3):313–420, 2010.

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

[56] Kokouvi Hounkanli and Andrzej Pelc. Asynchronous broadcasting with bivalent beeps. In *Proceedings of the 23rd International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, 2016. To appear.

[57] Bojun Huang and Thomas Moscibroda. Conflict resolution and membership problem in beeping channels. In *Proceedings of the 27th International Symposium on Distributed Computing (DISC)*, volume 8205 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2013.

[58] Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990.

[59] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[60] 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.

[61] Tomasz Jurdziński and Dariusz R. Kowalski. Distributed randomized broadcasting in wireless networks under the SINR model. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer US, 2014.

[62] Jörg Keller, Christoph W. Keßler, and Jesper Larsson Träff. *Practical PRAM Programming*. Wiley Series on Parallel and Distributed Computing. Wiley, 2001.

[63] Dariusz R. Kowalski and Adam Malinowski. How to meet in anonymous network. *Theoretical Computer Science*, 399(1-2):141–156, 2008.

[64] Evangelos Kranakis and Danny Krizanc. Distributed computing on anonymous hypercube networks. *Journal of Algorithms*, 23(1):32–50, 1997.

[65] Evangelos Kranakis, Danny Krizanc, and Flaminia L. Luccio. On recognizing a string on an anonymous ring. *Theory of Computing Systems*, 34(1):3–12, 2001.

[66] Evangelos Kranakis, Danny Krizanc, and Jacob van den Berg. Computing boolean functions on anonymous networks. *Information and Computation*, 114(2):214–236, 1994.

[67] Evangelos Kranakis and Nicola Santoro. Distributed computing on oriented anonymous hypercubes with faulty components. *Distributed Computing*, 14(3):185–189, 2001.

[68] 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.

[69] Lloyd Lim and A Park. Solving the processor identity problem in o (n) space. In *Parallel and Distributed Processing, 1990. Proceedings of the Second IEEE Symposium on*, pages 676–680. IEEE, 1990.

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

[71] Colin McDiarmid. On the method of bounded differences. In J. Siemons, editor, *Surveys in Combinatorics, 1989*, pages 148–188. Cambridge University Press, 1989.

[72] Yves Métivier, John Michael Robson, and Akka Zemmari. Analysis of fully distributed splitting and naming probabilistic procedures and applications. *Theoretical Computer Science*, 584:115–130, 2015.

[73] Othon Michail, Ioannis Chatzigiannakis, and Paul G. Spirakis. *New Models for Population Protocols*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2011.

[74] Othon Michail, Ioannis Chatzigiannakis, and Paul G. Spirakis. Naming and counting in anonymous unknown dynamic networks. In *Proceedings of the* 15*th Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 8255 of *Lecture Notes in Computer Science*, pages 281–295. Springer, 2013.

[75] Michael Mitzenmacher and Eli Upfal. *Probability and Computing*. Cambridge University Press, 2005.

[76] Arik Motskin, Tim Roughgarden, Primoz Skraba, and Leonidas J. Guibas. Lightweight coloring and desynchronization for networks. In *Proceedings of the* 28*th IEEE International Conference on Computer Communications (INFOCOM)*, pages 2383–2391, 2009.

[77] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[78] Saket Navlakha and Ziv Bar-Joseph. Distributed information processing in biological and computational systems. *Communications of the ACM*, 58(1):94–102, 2014.

[79] 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.

[80] Andrzej Pelc. Activating anonymous ad hoc radio networks. *Distributed Computing*, 19(5-6):361–371, 2007.

[81] John H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993.

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

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

[84] Baruch Schieber and Marc Snir. Calling names on nameless networks. *Information and Computation*, 113(1):80–101, 1994.

[85] Stefan Schmid and Roger Wattenhofer. Algorithmic models for sensor networks. In *Proceedings of the* 20*th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006.

[86] Alex Scott, Peter Jeavons, and Lei Xu. Feedback from nature: An optimal distributed algorithm for maximal independent set selection. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 147–156, 2013.

[87] Shang-Hua Teng. Space efficient processor identity protocol. *Information Processing Letters*, 34(3):147–154, 1990.

[88] 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.

[89] Uzi Vishkin. Using simple abstraction to reinvent computing for parallelism. *Communications of the ACM*, 54(1):75–85, 2011.

[90] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks: Part I-characterizing the solvable cases. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):69–89, 1996.

[91] Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proceedings of the* 18*th Symposium on Foundations of Computer Science (FOCS)*, pages 222–227. IEEE Computer Society, 1977.

[92] Jiguo Yu, Lili Jia, Dongxiao Yu, Guangshun Li, and Xiuzhen Cheng. Minimum connected dominating set construction in wireless networks under the beeping model. In *Proceedings of the 2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 972–980, 2015.