Wait-Free Implementations in Message-Passing Systems

Soma Chaudhuri *Maurice Herlihy* *Mark R. Tuttle*

Cambridge Research Laboratory

Technical Report Series

CRL 98/5

May 1998

Cambridge Research Laboratory

The Cambridge Research Laboratory was founded in 1987 to advance the state of the art in both core computing and human-computer interaction, and to use the knowledge so gained to support the Company's corporate objectives. We believe this is best accomplished through interconnected pursuits in technology creation, advanced systems engineering, and business development. We are actively investigating scalable computing; mobile computing; vision-based human and scene sensing; speech interaction; computer-animated synthetic persona; intelligent information appliances; and the capture, coding, storage, indexing, retrieval, decoding, and rendering of multimedia data. We recognize and embrace a technology creation model which is characterized by three major phases:

Freedom: The life blood of the Laboratory comes from the observations and imaginations of our research staff. It is here that challenging research problems are uncovered (through discussions with customers, through interactions with others in the Corporation, through other professional interactions, through reading, and the like) or that new ideas are born. For any such problem or idea, this phase culminates in the nucleation of a project team around a well articulated central research question and the outlining of a research plan.

Focus: Once a team is formed, we aggressively pursue the creation of new technology based on the plan. This may involve direct collaboration with other technical professionals inside and outside the Corporation. This phase culminates in the demonstrable creation of new technology which may take any of a number of forms - a journal article, a technical talk, a working prototype, a patent application, or some combination of these. The research team is typically augmented with other resident professionals—engineering and business development—who work as integral members of the core team to prepare preliminary plans for how best to leverage this new knowledge, either through internal transfer of technology or through other means.

Follow-through: We actively pursue taking the best technologies to the marketplace. For those opportunities which are not immediately transferred internally and where the team has identified a significant opportunity, the business development and engineering staff will lead early-stage commercial development, often in conjunction with members of the research staff. While the value to the Corporation of taking these new ideas to the market is clear, it also has a significant positive impact on our future research work by providing the means to understand intimately the problems and opportunities in the market and to more fully exercise our ideas and concepts in real-world settings.

Throughout this process, communicating our understanding is a critical part of what we do, and participating in the larger technical community—through the publication of refereed journal articles and the presentation of our ideas at conferences–is essential. Our technical report series supports and facilitates broad and early dissemination of our work. We welcome your feedback on its effectiveness.

Robert A. Iannucci, Ph.D.
Director

Wait-Free Implementations in Message-Passing Systems

Soma Chaudhuri
Department of Computer Science
Iowa State University
Ames, IA 50011
chaudhur@cs.iastate.edu

Maurice Herlihy
Computer Science Department
Brown University
Providence, RI 02912
herlihy@cs.brown.edu

Mark R. Tuttle
DEC Cambridge Research Lab
One Kendall Square, Bldg 700
Cambridge, MA 02139
tuttle@crl.dec.com

May 1998

Abstract

We study the round complexity of problems in a synchronous, message-passing system with crash failures. We show that if processors start in order-equivalent states, then a logarithmic number of rounds is both necessary and sufficient for them to reach order-inequivalent states. These upper and lower bounds are significant because they establish a complexity threshold below which no nontrivial problem can be solved, but at which certain nontrivial problems do have solutions.

This logarithmic lower bound implies a matching lower bound for a variety of decision tasks and concurrent object implementations. In particular, we examine two nontrivial problems for which this lower bound is tight: the *strong renaming* task, and a wait-free *increment register* implementation. For each problem, we present a nontrivial algorithm that halts in $O(\log c)$ rounds, where c is the number of participating processors.

CRL Technical reports are available on the CRL's web page at
http://www.crl.research.digital.com.

Digital Equipment Corporation
Cambridge Research Laboratory
One Kendall Square, Building 700
Cambridge, Massachusetts 02139 USA

1 Introduction

In a synchronous, message-passing system with crash failures, a computation proceeds in a sequence of *rounds*: each processor sends messages to the others, receives all the messages sent to it, and performs an internal computation. At any point in the computation, a processor may *crash*: it stops and sends no more messages. This model is one of the most thoroughly-studied models in the distributed computing literature, partly because it is so easy to describe, and partly because the behaviors exhibited by this model are a subset of the behaviors exhibited by almost any other model of computation, which means that lower bounds in this model usually extend to other models.

We investigate the time needed to solve nontrivial problems in this model. Loosely speaking, a nontrivial problem is one in which at least two processors must perform different actions, which is a kind of "symmetry breaking." In the well-known *consensus* problem [PSL80, LSP82, FLP85], each processor starts with an input value, and all nonfaulty processors must halt after agreeing on the input value of one of the processors. Consensus breaks symmetry by requiring one processor to choose its input value, and the rest to discard theirs. Consensus requires a linear number of rounds to solve [FL82], but it can be used to solve almost any other nontrivial problem [Lam78, Lam89, Sch87, Her91b], so solving these nontrivial problems never takes longer than consensus. We want to know exactly how quickly these nontrivial problems can be solved.

Solving a nontrivial problem requires causing two processors to perform different actions. Speaking informally, if processors start in equivalent states and follow the same deterministic protocol, then as long as they remain in equivalent states, they will continue to perform the same actions. We can therefore equate the number of rounds needed to reach inequivalent states with a threshold below which no nontrivial problem has a solution. Surprisingly, we can show that there do exist nontrivial problems that become solvable at exactly this threshold.

What does it mean for two processors to be in equivalent states? Each processor begins execution with a unique identifier (called its *id*) taken from a totally-ordered set. We assume that processors can test ids for equality and order: given ids p and q, a processor can test whether $p = q$, $p \leq q$, and $p \geq q$. We say that two processor states are *order-equivalent* [FL87] if they cannot be distinguished by the order of the ids appearing within them. More specifically, the two states must have the same structure, and the order of any two ids appearing in one state must be the same as the order of the ids appearing in the corresponding positions of the other state. For example, if each processor's initial state contains its id and nothing else, then all initial states are trivially order-equivalent because there are no pairs of ids to compare within an initial state. A protocol is *comparison-based* if the comparison operations are the *only* operations applied to process ids. Clearly, comparison-based protocols cannot distinguish between order-equivalent states.

We restrict our attention to comparison-based protocols. This restriction to protocols that can only compare processor ids for order is reasonable in systems where there are many more processor ids than there are processors, or in systems where there is a very large pool of potential participants, of which only an unpredictable subset actually participates in the protocol. In such systems, there may be no effective way to enumerate all possible processor ids, and no way to tell whether there exists a processor with an id between two other ids. Most significant, since there are so many possible ids, it is not feasible to use processor ids as indices into data structures as is frequently done in implementations of objects like atomic registers (see [SP89]).

This paper's first principal contribution is a proof that any comparison-based protocol for c processors has an execution in which all processors remain in order-equivalent states for $\Omega(\log c)$ rounds. As a result, any problem that requires breaking order-equivalence also requires $\Omega(\log c)$ rounds. Although the proof is elementary, this logarithmic lower bound is "universal" for this model in the sense that it is difficult to conceive of a nontrivial problem that can be solved without breaking order-equivalence. This bound is tight: we give a protocol that forces any two processors into order-inequivalent states within $O(\log c)$ rounds.

This paper's second principal contribution is to show that there exist nontrivial problems that do have solutions with $O(\log c)$ round complexity. The existence of these problems implies that the synchronous message-passing model undergoes a kind of "phase transition" at a logarithmic number of rounds. Below this threshold, it is impossible to solve any problem that requires different processors to take different actions. Beyond this threshold, however, solutions do exist to nontrivial problems.

We consider two classes of problems: decision tasks, and concurrent objects. A *decision task* is a problem in which each processor begins execution with a private input value, runs for some number of rounds, and then halts with a private output value satisfying problem-specific constraints. Consensus is an example of a decision task. By contrast, a *concurrent object* is a long-lived data structure that can be simultaneously accessed by multiple processors. A concurrent object implementation is *wait-free* if any nonfaulty processor can complete any operation on the object in a finite number of steps, even if other processors crash at arbitrary points in their protocols. A shared FIFO queue is an example of a concurrent object.

Strong renaming is a decision task in which processors start with input bits indicating whether to participate in the protocol, and participating processors must choose unique names in the range $1 \dots c$, where c is the number of participating processors. A weaker form of this task has been extensively studied in asynchronous models [ABND+87, ABND+90, HS93]. Any protocol for strong renaming can be used to break order-equivalence, so a logarithmic lower bound is immediate. This bound is tight: we give a nontrivial protocol that solves strong renaming in $O(\log c)$ rounds.

Our lower bound on order-equivalence also translates into a lower bound on a variety of wait-free concurrent object implementations. For example, an *increment register* is a concurrent object consisting of an integer-valued register with an *increment* operation that atomically increments the register and returns its previous value. Because we can use an increment register to break order-equivalence, the $\Omega(\log c)$ lower bound for order-inequivalence translates directly into an $\Omega(\log c)$ lower bound on any wait-free implementation of the *increment* operation. This bound is also tight: we give a nontrivial wait-free increment register implementation in which each *increment* halts in $O(\log c)$ rounds, where c is the number of concurrently executing increment operations.

Our increment register construction is interesting in its own right, since it is based on our optimal solution to the strong renaming problem. In general, implementing long-lived objects is inherently more difficult than solving decision tasks. A decision task is solved once, while operations can be invoked on an object repeatedly. Even worse, processors solving a decision task start together, while processors invoking operations on an object can arrive at different and unpredictable times. The major technical difficulty in our register construction is how to guarantee that increment operations starting at different times do not interfere.

The rest of this paper is organized as follows. In Section 2 we define our model of computation. In Section 3, we define formally what we mean by order-equivalence of processors and present the matching $\log c$ lower and upper bounds for the problem. In Section 4, we define the strong renaming problem. We then reduce the problem of eliminating order-equivalence to the strong renaming problem, thus obtaining the $\log c$ lower bound for strong renaming. We then show that this bound is actually tight with an efficient strong renaming algorithm. In Section 5 we define concurrent objects and their implementation. We then reduce the problem of eliminating order-equivalence to the problem of implementing several concurrent objects, thus obtaining the same $\log c$ lower bound on the complexity of these concurrent objects. Finally we give our optimal wait-free implementation of an increment register, based on the strong renaming algorithm. We close with a discussion of some open problems in Section 6.

2 Model

Our model of computation is a synchronous, message-passing model with crash failures. It is similar to the models used in a number of other papers [MT88, HM90, HF89].

A system consists of n unreliable *processors* $p_1, \ldots, p_n$ and an external *environment* e. We use n to denote the total number of processors in the system, and c to denote the number of these processors that actually participate in a protocol like strong renaming or access a concurrent object like an increment register. Each processor has a unique processor id taken from some totally-ordered set of processor ids. Each processor can send a message to any other processor and to the environment. The environment can send to any processor a message taken from some set of messages (including $\perp$ to denote "no input"). Each processor has access to a *global clock* that starts at 0 and advances in increments of 1. Computation proceeds in a sequence of *rounds*, with round k lasting from time $k-1$ to time k on the global clock. In each round, each processor receives some message (possibly $\perp$) from the environment, then it sends messages to other processors (including itself) and the environment, then it receives the messages sent to it by processors in that round, and then it changes state and enters the next round. Communication is reliable: a message sent in a round is guaranteed to be delivered in that round. Processors are not reliable: a processor can crash or fail at any time by just halting in the middle of a round after sending some subset of its messages for that round.

A *global state* is a tuple $(s_1, \ldots, s_n, s_e)$ of local states, one local state s_i for each processor p_i and one local state s_e for the environment. The *local state* for processor p_i includes the time on the global clock, its processor id, the history of messages it has received from the environment, and the history of messages it has received from other processors. The local state for the environment may contain similar information, but it certainly contains the sequence of messages it has sent to processors in the system, the pattern of failures exhibited by processors, and any other relevant information that cannot be deduced from processors' local states. An *execution* e is an infinite sequence $g_0 g_1 \ldots$ of global states, where g_i is the global state at time i.

Processors follow a deterministic *protocol* that determines what messages to send to processors and the environment during a round as a function of its local state. A processor follows its protocol in every round, except that a processor may *crash* or *fail* in the middle of a round. If p_i fails in round k, then it sends all messages in rounds $j < k$ as required by the protocol, it sends a proper subset of its messages in round k, and it sends no messages in rounds $j > k$. A processor is considered *faulty* in an execution if it fails in some round of that execution, and *nonfaulty* otherwise.

Without loss of generality, we can assume that processors follow a *full-information protocol* in which processors broadcast their entire local state to every processor (including itself) in every round, and apply a *message function* to their local states to determine what message to send to the environment in every round. Given the state of a processor in the full-information protocol, this state contains enough information for us to compute the processor's state at the corresponding point of any other protocol [FL87, MT88].

Also without loss of generality, since processors broadcast their entire local state in every round, we can assume that the *local state* of a processor is a LISP S-expression defined as follows. Let us fix some totally-ordered set $\mathcal{I}$ of processor ids, some set $\mathcal{S}$ of initial states, some set $\mathcal{M}$ of messages from the environment (including $\perp$), and some set $\mathcal{N}$ of messages from the processors to the environment. The initial state for a processor with processor id p starting in initial state s with initial input m from the environment is written $(p\ \ m\ \ s)$. Later states are written $(p\ \ m\ \ m_0\ \ m_1\ \ \ldots\ \ m_k)$, where p is the processor id, m is the input received from the environment at the start of the current round, and $m_0 \ldots m_k$ is the set of messages received from processors during the last round (including p itself) sorted by processor id. The messages m_i are themselves S-expressions representing the states of the sending processors at the start of the round, including p's local state. Notice that while processors send S-expressions to each other, they still send messages from a fixed set to the environment: the message function maps a processor's local state to a message in $\mathcal{N}$ that the processor sends to the environment in that round. Again, we can assume processor states have such a special representation since from such a description of a processor's state we can reconstruct the value of every variable v appearing in the actual state [FL87, MT88]. All of the protocols in this paper are stated using an Algol-like notation for the sake of convenience, but their translation into this model is straight-forward.

For any given protocol, an execution of the protocol is completely determined by the processor ids, the initial states, the inputs received from the environment, and the pattern of processor failures during the execution. We define an *environment graph* to be an infinite graph that records this information [MT88]. We define an environment graph to be a grid with n vertices in the vertical axis labeled with processor names $p_1, \ldots p_n$ — denoting physical processors and not their processor ids — and with a countable number of vertices in the horizontal axis labeled with times $0, 1, 2, \ldots$. The node representing processor p at time i is denoted by the pair $\langle p, i\rangle$. Given any pair of processors p and q and any round i, there is an edge from $\langle p, i-1\rangle$ to $\langle q, i\rangle$ if p successfully sends a message to q in round i, and the edge is missing otherwise. Each node $\langle p, i\rangle$ is labeled with the input received from the environment by processor p at time i. In addition, each node $\langle p, 0\rangle$ is labeled with p's processor id and initial state. Since processors fail by crashing, an environment graph must satisfy the following property: if k is the least integer such that an edge is missing from $\langle p, k\rangle$, then no edges are missing from $\langle p, j\rangle$ for all $j < k$ and all edges are missing from $\langle p, j\rangle$ for all $j > k$. An environment graph must also satisfy the property that every processor starts with a unique processor id. Given a set S of processor initial states, a set I of processor ids, and a set M of environment messages, we define $\mathcal{E}(S, I, M)$ to be the set of all such environment graphs labeled with initial states in S, processor ids in I, and messages from the environment in M.

We define the *local state* of the environment at time k to be the finite prefix of an environment graph describing the processor inputs and failures at times 0 through k, and we require that the local states of the environment in an execution be prefixes of the same environment graph. We define an *environment* to be a set of environment graphs. We will typically consider environments of the form $\mathcal{E}(S, I, M)$, or simple restrictions of such environments. For example, in the context of decision problems like consensus, we might consider an environment in which each processor receives a message from the environment (the processor's input bit) at time 0 and at no later time. Given a protocol P and an environment $\mathcal{E}$, we define $P(\mathcal{E})$ to be the set of all executions of P in the context of the environment graphs in $\mathcal{E}$.

3 Order-equivalence

In this section, we show that a logarithmic number of rounds is a necessary and sufficient amount of time for comparison-based protocols to reach order-inequivalent states. We begin with the definitions of comparison-based protocols and order-equivalent states. Both definitions are based on the assumption that the set of processor ids is a totally-ordered set, meaning that it is possible to test processor ids for relative order, but that it is not possible to examine the structure of ids in any more detail.

Informally, two states are order-equivalent if they cannot be distinguished by comparing the processor ids that appear within them. Remember that a processor state is represented by an S-expression in our model. A processor's initial state is written as $(p\ m\ s)$, where p is a processor id, m is a message from the environment, and s is an initial state. Later states are written $(p\ m\ m_0\ m_1\ \ldots\ m_k)$, where p is a processor id, m is a message from the environment, $m_0 \ldots m_k$ is some set of messages (S-expressions representing local states) received from processors during the last round and all sorted by processor id. Loosely speaking, two processor states s and s' are equivalent if (i) they are structurally equivalent S-expressions, (ii) initial states from corresponding positions in s and s' are identical, (iii) messages from the environment from corresponding positions in s and s' are identical, and (iv) if two ids p and q taken from s satisfy $p \leq q$, then the two ids p' and q' taken from corresponding positions of s' satisfy $p' \leq q'$. Intuitively, a processor cannot distinguish equivalent states simply by comparing processor ids: if a processor tries to learn something about its local state by sequentially testing pairs of ids appearing within that state for relative order, then this sequence of tests will yield the same results when applied to any other equivalent state.

Formally, a one-to-one function ϕ from one totally ordered set to another is *order-preserving* if $p \le q$ implies $\phi(p) \le \phi(q)$. Any such ϕ can be extended to S-expressions representing processor states by defining $\phi(p\ m\ s) = (\phi(p)\ m\ s)$ and

$$\phi(p\ m\ m_0\ \ldots\ m_k) = (\phi(p)\ m\ \phi(m_0)\ \ldots\ \phi(m_k)).$$

Two processor states are *order-equivalent* if there exists an order-preserving function ϕ mapping one to the other, and *order-inequivalent* otherwise. A protocol is a *comparison-based* protocol if the message function choosing the message in $\mathcal{N}$ that a processor is to send to the environment maps order-equivalent states to the same message.

3.1 Lower bound

First let us prove that every comparison-based protocol has an execution in which processors remain in order-equivalent states for a logarithmic number of rounds.

Given a protocol P, the larger the environment $\mathcal{E}$ — the more environment graphs $\mathcal{E}$ contains — the larger the set $P(\mathcal{E})$ of executions of P in this environment, and the more likely the set $P(\mathcal{E})$ contains a long execution. To make our lower bound as widely applicable as possible, we now define the smallest environment $\mathcal{F}$ for which we can prove the existence of the long execution. A processor is *active in round r* in an environment graph (or an execution) if it sends at least one message in round r, and a processor is *active* if it is active in any round (and, in particular, if it is active in round 1). Given a set S of processor initial states, a set I of processor ids, and a set M of environment messages, define $\mathcal{F}(S, I, M)$ to be the subset of all environment graphs in $\mathcal{E}(S, I, M)$ satisfying the condition that (i) each active processor starts with the same initial state from S and the same environment message from M, and (ii) each active processor receives $\perp$ (representing no input) from the environment at every time after time 0.

In such an environment, the long executions of a comparison-based protocol are the ones in which the processors fail according to a *sandwich failure pattern* in every round. This failure pattern is defined as follows. Suppose processors with ids $q_1, \ldots, q_u$ have not failed at the start of a round, and suppose $u = 3v + 1$ for some integer v. (The sandwich failure pattern can always fail the one or two processors with lowest ids at the beginning of the round and pretend they don't exist.) The sandwich failure pattern causes the v processors $q_1, \ldots, q_v$ with the lowest ids and the v processors $q_{2v+2}, \ldots, q_{3v+1}$ with the highest ids to fail in the following way: each processor $q_{v+j} \in \{q_{v+1}, \ldots, q_{2v+1}\}$ receives messages only from processors $q_j, \ldots, q_{2v+j}$. Notice that each such processor q_{v+j} sees $2v + 1$ active processors, and sees its rank in this set of active processors as $v + 1$. Notice also that the active processors after a round of the sandwich failure pattern is always a consecutive subsequence of processors from the middle of the sequence of active processors at the beginning of the round. Using this failure pattern, we can prove our lower bound:

Proposition 1: Let P be a comparison-based protocol, and let $\mathcal{E}$ be an environment containing an environment of the form $\mathcal{F}(S, I, M)$. For every $c \leq n$, there is an execution in $P(\mathcal{E})$ with c active processors in which the nonfaulty processors remain order-equivalent for $\Omega(\log_3 c)$ rounds.

Proof: Let G be an environment graph in $\mathcal{F}(S, I, M) \subseteq \mathcal{E}$ with the sandwich failure pattern and with c active processors with ids $q_1, \ldots, q_c$. Each active processor starts with the same initial state $s \in S$ and the same environment input $m \in M$, and receives $\perp$ from the environment at all times after time 0. Notice that the sandwich failure pattern fails roughly $2/3$ of the active processors, leaving roughly $1/3$ remaining active in the next round. A simple analysis shows that if $\ell < \log_3 c$, then the number $3v + 1$ of processors remaining at the end of round ℓ is at least four. We claim that if $\ell < \log_3 c$, then after ℓ rounds of the sandwich failure pattern the states s_i and s_j of processors q_i and q_j are related by an order-preserving function ϕ_{j-i} defined as follows. For all integers d, the function $\phi_d(q_i) = q_{i+d}$ is defined for $1 \leq i \leq c - d$ when $d \geq 0$ and for $-d + 1 \leq i \leq c$ when $d < 0$. We claim that $\phi_{j-i}(s_i) = s_j$. We proceed by induction on ℓ.

The result is immediate for $\ell = 0$ since each q_i's initial state is $(q_i\ m\ s)$, and

$$\phi_{j-i}(q_i\ m\ s) = (\phi_{j-i}(q_i)\ m\ s) = (q_j\ m\ s).$$

For $\ell > 0$, suppose the hypothesis is true for $\ell - 1$. Consider the $3v + 1$ processors that are active in round ℓ. Since the active processors at the beginning of round ℓ are a consecutive subsequence of $q_1, \ldots, q_c$, suppose they are $q_{a+1}, \ldots, q_b$. By the induction hypothesis for $\ell - 1$, the states $S_{a+1}, \ldots, S_b$ these processors have at the beginning of the round are related by $\phi_{j-i}(S_i) = S_j$. Notice that at the end of the round, due to the sandwich failure pattern in that round, the active processors are $q_{a+v+1}, \ldots, q_{b-v}$, and that the state of processor q_{a+v+i} is

$$(q_{a+v+i} \perp S_{a+i}\ \ldots\ S_{a+2v+i})$$

and the state of processor q_{a+v+j} is

$$(q_{a+v+j} \perp S_{a+j}\ \ldots\ S_{a+2v+j})$$

It is easy to see that ϕ_{j-i} maps the state of q_{a+v+i} to q_{a+v+j}, as desired. $\square$

3.2 Order-equivalence elimination algorithm

Now let us prove that this lower bound is tight. There is a simple algorithm that forces all processors into order-inequivalent states in a logarithmic number of rounds:

Proposition 2: There exists a protocol that leaves all nonfaulty processors in order-inequivalent states after $O(\log c)$ rounds, where c is the number of active processors in the execution.

Proof: Here is a comparison-based algorithm that causes nonfaulty processors to choose distinct sequences of integers after a logarithmic number of rounds. In each round, each processor broadcasts its id and the sequence of integers constructed so far. Two processors *collide* in a round if they broadcast identical sequences in that round. In round 1, a processor with id p broadcasts $(p,\ \epsilon)$, and hence all processors collide initially with the empty sequence ϵ. Let $(p,\ i_1\ \ldots\ i_{k-1})$ be the message p broadcast at round k, and let i_k be the number of processors with ids less than p that p hears from and that collide with p in round k. In round $k+1$, p broadcasts $(p,\ i_1\ \ldots\ i_k)$. Each processor halts when it does not hear from any colliding processor. As an example, suppose that p_1 fails in round 1 by sending a message to p_3 but not to p_4. Then p_3 receives $(p_i,\ \epsilon)$ from p_1, p_2, p_3, p_4 and p_4 receives $(p_i,\ \epsilon)$ from p_2, p_3, p_4, so both p_3 and p_4 will consider its rank in the processors it hears from in round 1 to be 3, both will broadcast $(p_i, 3)$ in round 2, and both will collide again at the end of round 2.

We claim that the size of maximal sets of colliding processors must shrink by approximately half with each round, yielding an $O(\log c)$ running time. Two processors that broadcast different sequences continue to do so, so the set of processors that collide with p at round k is a subset of the processors that collided with p at earlier rounds. Consider a maximal set S of processors that collide in round k; that is, a set of ℓ processors that do not fail in round $k-1$ and broadcast the same sequence $i_1 \ldots i_{k-1}$ in round k. Let p be the lowest processor in that set, and let q be the highest. Since processors in S do not fail in round $k-1$, processor q must hear from each of the ℓ processors in S in round $k-1$. Since these processors collide with q in round k, they must collide with q in round $k-1$ as well, so q must count at least $\ell-1$ colliding processors with lower ids that broadcast the sequence $i_1 \ldots i_{k-2}$ in round $k-1$. It follows that $i_{k-1} \geq \ell - 1$ for processor q. Since p and q collide at round k, they broadcast the same value for i_{k-1} in round k, so $i_{k-1} \geq \ell-1$ for processor p as well. Therefore, processor p must see at least $\ell-1$ colliding processors with lower ids broadcasting the sequence $i_1 \ldots i_{k-2}$ in round $k-1$, none of which are in S (since p is the processor with smallest id in S). Hence at least $2\ell-1$ processors broadcast the sequence $i_1 \ldots i_{k-2}$ in round $k-1$ and collide with p and q in round $k-1$, implying that the number of processors colliding with p and q has shrunk from at least $2\ell-1$ to ℓ in one round, which is a reduction by a factor of 2. $\square$

Since the logarithmic bounds are tight, these results show that the logarithmic bounds are the best possible bounds that can be obtained in this model using the notion of order-equivalence. In the remainder of this paper, we will show that this logarithmic lower bound can be used to prove logarithmic lower bounds for decision problems like strong renaming and concurrent objects like increment registers. Since this logarithmic bound is tight for order-equivalence, these results show, for example, that if operations on objects such as stacks or queues require more than a logarithmic number of rounds, then this additional complexity cannot be an artifact of the comparison model, but must somehow be inherent in the semantics of the objects themselves.

4 Decision problems

We can use the lower bound on order equivalence to prove lower bounds for decision problems. For example, consider the problem of *strong renaming* defined as follows. Each processor has a unique id taken from a totally-ordered set of ids. At the start of the protocol, each processor is in a distinguished initial state and receives a single bit from the environment, either 0 or 1 meaning "don't participate" or "participate," respectively. At the end of the protocol, each participating processor sends an integer to the environment. A protocol solves the strong renaming problem if each nonfaulty participating processor sends a distinct integer from $\{1, \ldots, c\}$ to the environment at the end of every execution in which at most $n-1$ processors fail, where $c \leq n$ is the number of participating processors.

4.1 Lower bound

The logarithmic lower bound for strong renaming follows quickly from the lower bound for order-inequivalence:

Proposition 3: Any comparison-based protocol for strong renaming requires $\Omega(\log_3 c)$ rounds of communication, where c is the number of participating processors.

Proof: Let P be a comparison-based protocol for strong renaming. According to the definition of strong renaming, each processor has a unique id from a totally-ordered set I, each processor starts in the same initial state s, and each processor receives a bit in $\{0, 1\}$ from the environment. Consequently, the environment $\mathcal{E}$ for this problem includes the environment $\mathcal{F}(\{s\}, I, \{1\})$ consisting of environment graphs in which all active processors start with the same state s and all active processors start with the same participation bit 1. In this environment, the active processors are precisely the participating processors.

Since each processor terminates by sending a different integer to the environment, and since the message function of a comparison-based protocol — the function computing the messages processors send to the environment — must be the same in order-equivalent states, the processors must end the protocol in order-inequivalent states. By Proposition 1, for every $c \leq n$, there must be some execution of P in $P(\mathcal{E})$ in which the c active (and hence participating) processors are still order-equivalent after $\Omega(\log_3 c)$ rounds. Consequently, for every $c \leq n$, there must be some execution of P that requires $\Omega(\log_3 c)$ rounds. □

Lower bounds for other decision problems like order-preserving renaming can also be proven using the same technique.

4.2 Strong renaming algorithm

The logarithmic lower bound for strong renaming is tight, because there is a simple algorithm solving strong renaming in a logarithmic number of rounds. The algorithm is given in Figure 1.

The basic idea is that if a processor p hears of 2^b other participating processors, then it chooses a b-bit name for itself one bit at a time, starting with the high-order bit and working down to the low-order bit. Every round, p sends an interval I containing the names it is willing to choose from. On the first round, when the processor has not yet chosen any of the leading bits in its name, it sends the entire interval $[1, 2^b]$. It sets its high-order bit to 0 if it finds it is in the bottom half of the set of processors it hears from

define $\mathit{rank}(s, S) = |\{s' \in S : s' \leq s\}|$
define $\mathit{bot}(S) = \{s \in S : \mathit{rank}(s, S) \leq |S|/2\}$
define $\mathit{top}(S) = S - \mathit{bot}(S)$
define $\mathit{bot}(S, k) = \{s' \in S : \mathit{rank}(s', S) \leq k/2\}$

broadcast p
$\mathcal{P} \leftarrow \{p' : p' \text{ received}\}$
$b \leftarrow \lceil \log |\mathcal{P}| \rceil$
$I \leftarrow [1, 2^b]$

repeat
    broadcast (p, I)
    $\mathcal{I} \leftarrow \{I' : (p', I') \text{ received and } I \cap I' \neq \emptyset\}$
    $\mathcal{P} \leftarrow \{p' : (p', I') \text{ received and } I \cap I' \neq \emptyset\}$
    if $I' \subseteq I$ for every $I' \in \mathcal{I}$ then
        if $p \in \mathit{bot}(\mathcal{P}, |I|)$
            then $I \leftarrow \mathit{bot}(I)$
            else $I \leftarrow \mathit{top}(I)$
until $|I'| = 1$ for all $I' \in \{I' : (p', I') \text{ received}\}$

return a, where $I = [a, a]$

Figure 1: A $\log c$ renaming protocol $\mathcal{A}$ for processor p.

interested in names from the interval $[1, 2^b]$, and to 1 if it finds itself in the top half. In the first case it sends the interval $[1, 2^{b-1}]$, and in the second it sends $[2^{b-1}+1, 2^b]$. In order to make an accurate prediction of the behavior of processors interested in names in its interval I, however, it must wait until every processor interested in names in I is interested *only* in names in I before choosing its bit and splitting its interval in half; that is, it must wait until its interval I is maximal among the intervals intersecting I. Continuing in this way, the processor chooses each bit in its name, and continues broadcasting its name until all processors have chosen their name.

There are a few useful observations about the intervals processors send during this algorithm. The first is that if processor p sends the interval I_k during round k, then $I_k \supseteq I_{k'}$ for all later rounds $k' \geq k$. The second is that each interval I_k is of a very particular form; namely, every interval sent during an execution of $\mathcal{A}$ is of the form $[d2^j + 1, d2^j + 2^j]$ for some constant d. This is easy to see since the first interval I_2 a processor sends (in round 2) is of the form $[1, 2^b]$, and every succeeding interval I_k is either I_{k-1} or of the form $\mathit{top}(I_{k-1})$ or $\mathit{bot}(I_{k-1})$ (as defined in Figure 1). We say that an interval I is a *well-formed* interval if it is of the form $[d2^j + 1, d2^j + 2^j]$ for some constant d. It is easy to see that any two well-formed intervals are either distinct, or one is contained in the other. Notice that every well-formed interval I is properly contained in a unique minimal, well-formed interval $\hat{I} \supset I$. Furthermore, either $I = \mathit{top}(\hat{I})$ or $I = \mathit{bot}(\hat{I})$, and it is the low-order bit of the constant d that tells us which is the case. We define the operator $\hat{I}$ that maps a well-formed interval I to the unique minimal, well-formed interval $\hat{I}$ properly containing I.

In every round of the algorithm, a processor p computes the set $\mathcal{P}$ of processors with intervals I' intersecting its current interval I. These processors in $\mathcal{P}$ are the processors p is competing with for names in I. When p sees that its interval I is a maximal interval (that is, all intervals I' received by p that intersect I are actually contained in I), processor p adjusts its set I to either $bot(I)$ or $top(I)$. Our first lemma essentially says that when p replaces I with $bot(I)$ or $top(I)$, there are enough names in I to assign a name from I to every competing processor. Furthermore, this lemma shows that when a processor's interval reduces to a singleton set, then this processor no longer has any competitors for that name.

Lemma 4: Suppose p sends interval I during round $k \geq 2$. If P is the set of processors sending an interval $I' \subseteq I$ during round k, then $|P| \leq |I|$.

Proof: We consider two cases: $I = bot(\hat{I})$ and $I = top(\hat{I})$.

First, suppose $I = bot(\hat{I})$. Consider the greatest processor q (possibly p itself) in P. Processor q sent an interval $J_k \subseteq I$ to p in round k, so consider the first round $\ell \leq k$ in which q sent some interval $J \subseteq I$ to any processor (and hence to p).

If $\ell = 2$, then J is of the form $[1, 2^b]$, where 2^b is an upper bound on the number of processors q heard from in round 1, and hence on the number of active processors in round $k \geq 2$, and therefore on $|P|$, the number of processors sending intervals contained in I in round k. It follows that $|P| \leq 2^b = |J| \leq |I|$.

If $\ell > 2$, then q sent $J \subseteq I$ in round ℓ, and q sent a larger interval $\hat{J} \not\subseteq I$ in round $\ell - 1$, since ℓ is the first round that q sent an interval contained in I. In fact, we must have $J = I$ and $\hat{J} = \hat{I}$, for if $J \subset I$ then $\hat{J} \subseteq I$ and ℓ is not the first round that q sent an interval contained in I, a contradiction. Let $\mathcal{P}$ be the set of processors sending an interval intersecting $\hat{I}$ to q in round $\ell - 1$. Since every processor $p' \in P$ sending an interval $I' \subseteq I$ to p in round k must also send an interval intersecting $\hat{I}$ to q in round $\ell - 1$, each of these processors must be contained in $\mathcal{P}$, and hence $P \subseteq \mathcal{P}$. Since q sent $\hat{I}$ in round $\ell - 1$ and $I = bot(\hat{I})$ in round ℓ, it must be the case that $q \in bot(\mathcal{P}, |\hat{I}|)$ at the end of round $\ell - 1$. Since $P \subseteq \mathcal{P}$ and since q is the greatest processor in P, it must be the case that $P \subseteq bot(\mathcal{P}, |\hat{I}|)$. It follows that $|P| \leq |\hat{I}|/2 = |I|$, as desired.

Now, suppose $I = top(\hat{I})$. The proof in this case is similar to the proof when $I = bot(\hat{I})$, except that q is now taken to be the *least* processor in P. □

Our second lemma shows that when a processor p selects an interval $I = [a, b]$, there are enough participating processors to use up the names $1, \ldots, a$. In particular, when p's interval becomes the singleton set $[a, a]$, then there are at least a participating processors, and hence a is a valid name for p to choose. We say that a processor *holds* an interval $[a, b]$ during a round if $[a, b]$ is its interval at the beginning of the round, and hence the interval it sends during that round (if it sends any interval at all).

Lemma 5: If $I = [a, b]$ is a maximal interval received by some processor p during round k, then there are at least $a - 1$ processors that either hold an interval $[a', b']$ with $b' < a$ during round k or fail to send to p in round k.

Proof: We proceed by induction on k.

Suppose $k = 2$. This is the first round that any processor sends any interval, so I must be of the form $[1, 2^b]$, and it is vacuously true that at least 0 processors fail to send to p in round 2. Now suppose $k > 2$, and suppose the induction hypothesis holds for $k' < k$.

Suppose $I = bot(\hat{I})$. Processor p received I in round k, so consider any processor q that sent I during round k, and consider the first round $k' \leq k$ in which q sent I. If $k' = 2$, then I is of the form $[1, 2^b]$, and it is vacuously true that 0 processors fail to send to p, so suppose $k' > 2$. In this case, q must send $\hat{I}$ during round $k' - 1$ and I during round k', and the fact that q splits down at the end of round $k' - 1$ implies that $\hat{I}$ is a maximal interval received by q during round $k' - 1$.

Since intervals I and $\hat{I}$ have the same lower bound a, the induction hypothesis for $k' - 1 \leq k - 1$ implies that there are at least $a - 1$ processors that either hold an interval $[a', b']$ with $b' < a$ during round $k' - 1$ or fail to send to q in round $k' - 1$. Since each processor that holds an interval $[a', b']$ in round $k' - 1$ must hold an interval $[a'', b'']$ contained in $[a', b']$ in round k or fail to send to p in round k, and since each processor that fails to send to q in round $k' - 1$ must fail to send to p in round k, it follows that there are at least $a - 1$ processors that either hold an interval $[a'', b'']$ with $b'' \leq b' < a$ in round k or fail to send to p in round k.

Suppose $I = top(\hat{I})$. Let q be the smallest processor ever sending I during the execution. The interval I is not the interval that q sent in round 2 — the first round in which any processor sends any interval — because in that round q sent an interval of the form $[1, 2^b]$, which is not of the form $top(\hat{I})$. Since I is a maximal interval received by p in round k, it follows that q must have sent the interval I for the first time in some round $k' \leq k$ and the interval $\hat{I} = [\hat{a}, b]$ in round $k' - 1$. Since q changed intervals between round $k' - 1$ and k', the interval $\hat{I}$ must be a maximal interval received by q in round $k' - 1$. By the induction hypothesis there are at least $\hat{a} - 1$ processors that either hold an interval $[a', b']$ with $b' < \hat{a}$ during round $k' - 1$ or fail to send to q in round $k' - 1$, and all of these processors must either hold an interval $[a', b']$ with $b' < \hat{a} < a$ in round k or fail to send to p in round k. Since q changed its interval from $\hat{I} = [\hat{a}, b]$ to $I = top(\hat{I}) = [a, b]$ at the end of round $k' - 1$, there are at least $a - \hat{a} = |\hat{I}|/2$ processors $q' < q$ sending an interval contained in $\hat{I}$ to q in round $k' - 1$. None of these processors q' sending an interval in $\hat{I} = [\hat{a}, b]$ to q in round $k' - 1$ could have been one of the $\hat{a} - 1$ processors that either held an interval $[a', b']$ with $b' < \hat{a}$ in round $k' - 1$ or failed to send to q in round $k' - 1$. All of these processors q' sending an interval in $\hat{I} = [\hat{a}, b]$ to q in round $k' - 1$ must either send an interval in $bot(\hat{I})$ to p in round k or fail to send to p in round k, since $I = top(\hat{I})$ is a maximal interval received by p in round k, and since $q' < q$ and we chose q to be the smallest processor ever sending I during the execution. It follows that at least $(\hat{a} - 1) + (a - \hat{a}) = a - 1$ processors hold an interval $[a', b']$ with $b' < a$ in round k or fail to send to p in round k, and we are done. □

Finally, since the algorithm terminates when every processor's interval is a singleton set, and since the size of the maximal interval sent during a round decreases by a factor of 2 in every round, it is easy to prove that the algorithm $\mathcal{A}$ terminates in $\log c$ rounds.

Lemma 6: The algorithm $\mathcal{A}$ terminates in $\log c + 2$ rounds, where c is the number of participating processors.

Proof: Consider an arbitrary execution of $\mathcal{A}$. For each round k, let ℓ_k be the size of the largest interval sent during round k.

Consider round $k = 2$. In this round, each processor p sends an interval of the form $[1, 2^b]$ where 2^b is the least power of two greater than or equal to the number of processors that processor p heard from in round 1. It follows that $\ell_2 = 2^b < 2c$, for some b, where c is the number of participating processors.

Consider any round $k > 2$ with $\ell_{k-1} > 1$. We claim that $\ell_k \leq \ell_{k-1}/2$. Consider any processor that holds an interval of size $\ell_{k-1} > 1$ at the start of round $k-1$, and hence sends this interval in round $k-1$. Since no interval sent in round $k-1$ is larger than ℓ_{k-1}, this processor must see that its interval is maximal at the end of round $k-1$ and split its interval in half for the next round. Since this is true for every processor sending an interval of size ℓ_{k-1} during round $k-1$, and every processor sending a smaller interval during round $k-1$ sends an interval of size at most $\ell_{k-1}/2$, it follows that all processors send an interval of size at most $\ell_{k-1}/2$ in round k, so $\ell_k \leq \ell_{k-1}/2$.

Since $\ell_2 < 2c$ and $\ell_k \leq \ell_{k-1}/2$, we have $\ell_k \leq \ell_2/2^{k-2} < 2c/2^{k-2}$. It follows that $\ell_k = 1$ within at most $k = \log c + 2$ rounds, at which time all intervals are of size 1 and the processors can halt. □

With these results, we are done:

Theorem 7: The algorithm $\mathcal{A}$ solves the strong renaming problem, and terminates in $\log(c) + 2$ rounds, where c is the number of participating processors.

Proof: First, by Lemma 6, all processors choose a name and terminate in $\log(c) + 2$ rounds, where c is the number of participating processors.

Second, the names chosen by processors are distinct. Suppose two processors p and p' chose the name a at the end of rounds k and $k' \geq k$, respectively. Processors p and p' must have sent the singleton set $I = [a, a]$ to all processors in rounds k and k', and intervals containing I in all preceding rounds. Since p could not have terminated in round k unless all intervals it received were singletons, both processors must have sent $I = [a, a]$ in round k. It follows by Lemma 4 that $2 \leq |I| = 1$, which is impossible.

Finally, names chosen are in the interval $[1, c]$, where c is the number of participating processors. Consider the processor p choosing the highest name a chosen by any processor, and consider the last round k in which p sends the singleton set $I = [a, a]$ and terminates, returning a. All intervals p receives in round k must therefore be singleton sets. This implies that I is a maximal interval received by p in round k. It follows by Lemma 5 that at least $a - 1$ processors hold intervals $[a', b']$ with $b' < a$ in round k or have failed, and hence that there are at least a participating processors. This implies that $c \geq a$ and all names are chosen in the interval $[1, c]$, as required. □

5 Wait-free objects

We can use the lower bound on order equivalence to prove lower bounds on the complexity of wait-free implementations of concurrent objects. We can also prove that this bound is tight in the case of a simple object called an increment register. The implementation that we describe is very similar to the strong renaming algorithm in the previous section. We start with some formal definitions.

An *object* is a data structure that can be accessed concurrently by all processors. It has a *type*, which defines the set of possible *values* the object can assume, and a set of *operations* that provide the only means to access or modify the object. A processor invokes an operation by sending an *invoke* message to the object, and the operation returns with a matching *response* message from the object. A *history* is a sequence of invoke/response messages. A *sequential history* is a history in which every invoke message is followed immediately by a matching response message, meaning that the operations are invoked sequentially one after another. In addition to a type, an object has a *sequential specification* which is a set of all possible sequential histories describing the sequential behavior of the object. For example, an *increment register* is just a register with an *increment* operation. The value of the register is a nonnegative integer. The *increment* operation atomically increments the value of the register and returns the previous value. The sequential behaviors for an increment register are the sequential histories of *increment* operations returning integer values in order, such as $0, 1, 2, \ldots$.

We are interested in concurrent implementations of such objects. To us, given an object O intended to be used by n processors $P_1, \ldots, P_n$, an implementation of O will be a collection of n processors $F_1, \ldots, F_n$ called *front ends* [Her91a] that process the invocations from $P_1, \ldots, P_n$ and return the responses from O. Intuitively, the front end F_i is just the procedure that processor P_i calls to invoke an operation on O. The front end F_i receives the invocations from P_i and sends the responses from O. In our model, since we are only concerned with the implementation of objects (and not their use), we assume that the front ends $F_1, \ldots, F_n$ are really the system processors $p_1, \ldots, p_n$. We assume that the invoking processors $P_1, \ldots, P_n$ are part of the external environment e, and we ignore them completely. With this in mind, we define a *history* of a system to be the history h obtained by projecting an execution of the system onto the subsequence of invoke/response messages appearing in the execution.

The specification of an object's concurrent behavior is defined in terms of its sequential specification. An object is *linearizable* [HW90] if each operation appears to take effect instantaneously at some point between the operation's invocation and response. Linearizability implies that operations on the object appear to be interleaved at the granularity of complete operations, and that the order of nonoverlapping operations is preserved. An implementation is said to be *wait-free* if no front end is blocked by the failure of other front ends. The precise definition of wait-free linearizable objects is well-known [HW90], so we will not repeat it here.

We assume that any wait-free, linearizable implementation of an object can be initialized to any value defined by the type of the object. Specifically, we assume that for every value v in the type of an object, there is an initial processor state s_v with the following property: if every processor begins in state s_v — with the possible exception of the processors failing immediately at time 0 — then every execution from this initial state is linearizable to a sequential history in which the operations in the history are invoked sequentially on a copy of the object initialized to the value v. This assumption is valid, for example, for all concurrent objects implemented using the technique of state machine replication [Lam78, Lam89, Sch87], which is the technique most commonly used in message-passing models like ours.

5.1 Lower bounds

Our lower bound on order-equivalence can be used to prove lower bounds for a number of concurrent objects. For example, an *ordered set* S is an object whose value is some subset of a totally ordered set T, with an $insert(a)$ operation that adds $a \in T$ to S and a $remove$ operation that removes minimum element $a \in S$ from S and returns a. As the next result shows, we can use the *remove* operation from any implementation of an ordered set to solve the order-equivalence problem, so the logarithmic lower bound on order-equivalence implies the same lower bound for the *remove* operation of an ordered set. Many interesting concurrent objects are special cases of an ordered set. For example, an ordered set's *remove* operation is just a special case of a stack's *pop*, a queue's *dequeue*, and a heap's *min*. Consequently, the next result implies a logarithmic lower bound for each of these operations as well.

Proposition 8: Given any comparison-based, wait-free, linearizable implementation of an ordered set, the *remove* operation requires $\Omega(\log_3 c)$ rounds of communication, where c is the number of concurrent invocations of the *remove* operation.

Proof: Consider any such implementation of the ordered set. Consider any value S of the ordered set containing at least n distinct values, where n is the number of processors in the system, and let s_S be the initial processor state with the property that if all processors start in state s_S, then the object is initialized to S. The environment $\mathcal{E}$ for an ordered set certainly includes the environment $\mathcal{F}(\{s_S\}, I, \{\mathit{remove}\})$ consisting of environment graphs in which all active processors start with the same state s_S and all active processors start with an invocation of the *remove* operation from the environment. In such an environment, the invoking processors are precisely the active processors.

Each processor terminates by removing a distinct value from the set and returning it to the environment by sending a distinct message to the environment. Since the message function of a comparison-based protocol — the function choosing the messages from $\mathcal{N}$ that processors send to the environment — must be the same in order-equivalent states, the processors must end the protocol in order-inequivalent states. By Proposition 1, for every $c \leq n$, there must be some execution of P in $P(\mathcal{E})$ in which the c active (and hence invoking) processors are still order-equivalent (and hence cannot terminate) after $\Omega(\log_3 c)$ rounds. $\square$

As another example, consider the *increment register* defined earlier in this section. The value of an increment register is just a nonnegative integer. The increment register provides an *increment* operation that atomically increments the value of the register and returns this new value. Since we can use the *increment* operation to solve the order equivalence problem, we can prove a logarithmic lower bound for the *increment* operation:

Proposition 9: Given any comparison-based, wait-free, linearizable implementation of an increment register, the *increment* operation requires $\Omega(\log_3 c)$ rounds of communication, where c is the number of concurrent invocations of the *increment* operation.

Proof: Consider any such implementation S of an increment register. Let s_0 be the initial processor state with the property that if every processor begins in state s_0, then the register is initialized to 0. The environment $\mathcal{E}$ for an increment register includes the environment $\mathcal{F}(\{s_0\}, I, \{\textit{increment}\})$ consisting of environment graphs in which all active processors start with the same state s_0 and all active processors start with an invocation of the *increment* operation from the environment. In this environment, the active processors are the incrementing processors.

Each processor terminates by returning a distinct value to the environment. Since the message function of a comparison-based protocol must send the same message to the environment in order-equivalent states, the processors must end the protocol in order-inequivalent states. By Proposition 1, for every $c \leq n$, there must be some execution of P in $P(\mathcal{E})$ in which the c active (and hence incrementing) processors are still order-equivalent (and hence cannot terminate) after $\Omega(\log_3 c)$ rounds. □

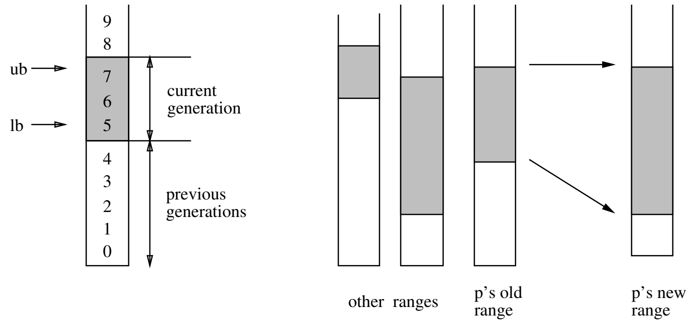
5.2 Increment register algorithm

In this section, we give the last major result of our paper: an optimal wait-free implementation of an increment register. It closely resembles our optimal strong renaming algorithm, and proves that the logarithmic lower bound is tight.

A processor p can invoke an increment operation multiple times in a single execution, and each invocation can take multiple rounds to complete. We refer to the set of increment operations invoked during round k as *generation k increments*, and we refer to the processors invoking these increments as *generation k processors*. We refer to the rounds of a generation as *phases*, and we number the phases of generation k starting with 0 so that phase ℓ of generation k occurs during round $k + \ell$.

Since a processor p can invoke the increment operation more than once, it identifies itself during generation k with an ordered pair $\langle p, k\rangle$ called its *increment processor id*. We assume each processor p maintains a set $IncSet$ of all the increment processor ids that it knows about, and continues to maintain this set in the background even when it is not actually performing an increment operation. Every round, it broadcasts this set to other processors, and merges the sets it receives from other processors into its own set. For notational simplicity, however, since the generation k will always be clear from context, we will frequently write p in place of $\langle p, k\rangle$. This set $IncSet$ can also be used to initialize the increment register. For the sake of simplicity, the implementation we give assumes the register is initialized to 0, but it can be initialized to any value as follows: in the initial processor state s_i in which the increment register has been initialized to the value i, the set $IncSet$ is initialized to contain phantom increment ids $\langle p, -i\rangle, \langle p, -i+1\rangle, \ldots \langle p, -1\rangle$ for some processor id p to simulate p's previously incrementing the register i times.

Understanding our implementation requires understanding the notions of *ranges*, *intervals*, *splitting*, and *chopping*, so let us begin with these concepts.



(a) Initial range. (b) Extending the range.

Figure 2: The range.

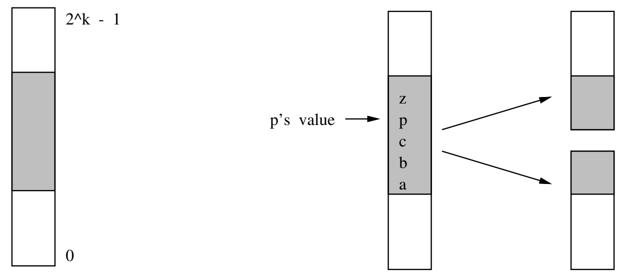
Ranges Our implementation has the property that increments in one generation are effectively isolated from increments in other generations, in the sense that increments in one generation can choose return values by communicating among themselves, ignoring increments in other generations. This isolation is achieved by partitioning the return values into ranges.

As illustrated in Figure 2, each processor p maintains a *range* $R = [R.lb, R.ub]$ of return values. Initially, using the set $IncSet$ of increment processor ids known to p, processor p sets its lower bound lb to the number of increments invoked by previous generations, and its upper bound ub to the total number of increments invoked by previous and current generations. Every phase, processor p exchanges ranges with other processors in its generation, and extends its range by dropping its lower bound to the smallest lower bound received from any of these processors.

Intuitively, by setting its initial lower bound to lb, processor p is reserving lower values $v < lb$ as return values for increments in previous generations recorded in $IncSet$. Later, if p hears that another processor q in the same generation set its initial lower bound to $lb' < lb$, then p knows some of these earlier increments have failed, so p ceases to reserve return values for them and drops its lower bound to lb'.

Our algorithm guarantees that if a nonfaulty processor sets its upper bound to ub, then all processors in all later generations set their initial lower bounds to $lb > ub$, so their lower bounds remain above ub forever. In this sense, the upper bounds of the nonfaulty processors partition the return values. Nonfaulty processors in different generations have disjoint ranges, allowing them to ignore each other once their initial ranges have been chosen.

Intervals and Splitting Given a range R of acceptable return values, however, p still has to choose one of them to return. To do so, we modify the fundamental idea in the optimal algorithm for strong renaming described in Section 4. The basic idea is that if the values in p's range R are b bits long, then p chooses a b-bit value from R one bit at a time, starting with the high-order bit and working down to the low-order bit. To



(a) Initial interval contains initial range. (b) Interval splits to the half containing p's value.

Figure 3: The interval.

implement this idea, processor p maintains an *interval* $I = [I.lb, I.ub]$ of return values that contains its range R (see Figure 3). The size of the interval is always a power of 2. Processor p's initial interval is the smallest interval of the form $[0, 2^k - 1]$ that contains p's initial range. During an increment, processor p repeatedly splits its interval in half until the interval contains a single value, and this is the value that p returns. It is easy to see that all of the intervals generated by p are of the form $[a2^k, a2^k+(2^k-1)]$ for some $b - k$ bit value a, and such intervals are called *well-formed* intervals. Intuitively, this interval represents the fact that p has chosen a as the high-order $b - k$ bits of its return value, but must still choose the low-order k bits.

The procedure that p uses to split its interval in half is important (see Figure 3). Every round, processor p exchanges intervals with other processors, and p maintains a set C of all processors sending p an interval intersecting its current interval I. The processors in C are p's *competitors* since they include the processors considering return values in p's range. To avoid returning the same value as one of its competitors, processor p attempts to predict what values its competitors will choose. To predict accurately, however, p must wait until I is maximal among the intervals received from its competitors; this means that p's competitors are considering only values in I. Once I is maximal, p assigns return values from its range to its competitors, starting at the bottom of its range and assigning values to competitors in order of increasing processor id. Eventually, p assigns a value v to itself. Processor p then replaces I with its top half $top(I)$ or its bottom half $bot(I)$—whichever half contains v—and then replaces its range R with the intersection of R and I. Continuing in this way every round, processor p's interval eventually contains a single value v, at which point p chooses v but continues exchanging its interval with other processors until all processors in its generation have chosen a value.

Chopping It is easy to see that the split operation is what gives rise to the algorithm's logarithmic nature: in any given round, a maximal interval is guaranteed to split in half, so the size of the maximal intervals decreases by a factor of 2 with every round. Unfortunately, this is logarithmic in the size of the initial interval, which can be as

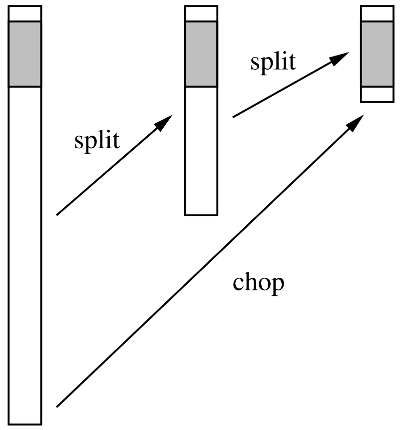


Figure 4: Chopping.

large as the total number of increments ever invoked, and we want the algorithm to run in time logarithmic in the number of concurrently executing increments. Fortunately, we can speed up the algorithm dramatically by introducing a new operation called a *chop*, illustrated in Figure 4. For example, if p's range R is just the top few values in its interval I, then it is clear that p is going to split up repeatedly for many rounds. We accelerate this splitting by allowing p to *chop* in a single round from I up to the smallest well-formed interval I' containing R. We say that p *chops* up in this case, and chopping down is similar. Since chopping is just an accelerated form of splitting, processor p must wait until I is maximal among the intervals received from its competitors before chopping. On the other hand, it is important that we do not allow p to split and chop in the same round: if p splits down and then immediately chops up to a smaller interval containing its new range, then it runs the risk of chopping away the bottom of its interval before learning that it can extend this range by dropping the lower bound, so it runs the risk of reaching a state in which its interval and range are too small to assign distinct values from its range to all of its competitors.

Algorithm With this, we have introduced the notions of ranges, intervals, splitting, and chopping, and we can turn our attention to the increment register implementation $\mathcal{I}$ itself. The main loop of the algorithm is given in Figure 5, the definitions of splitting and chopping are given in Figure 6, and the definitions of some initialization steps are given in Figure 7.

During the initial phases of generation k, an incrementing processor p starts by adding its increment processor id $\langle p, k\rangle$ to $IncSet$; it exchanges $IncSet$ with other processors and uses the result to choose its initial range R as described above; it exchanges R with other processors, extends R by dropping its lower bound as described above, and uses the result to choose its initial interval. In all later phases, processor p

```
begin /* a generation k increment by processor p */
  initialize(); /* add <p,k> to IncSet */
  phase0();    /* bcast IncSet, choose initial range R */
  phase1();    /* bcast R, extend R, and
                            choose initial interval I */

  repeat
    broadcast <p,R,I,lb>
    receive   <p',R',I',lb'> from generation k
                                          processors p'

    /* collect names and intervals of competitors */
    C <- {p':<p',R',I',lb'> received and I' intersects I}
    N <- {I':<p',R',I',lb'> received and I' intersects I}

    /* extend range by dropping the lower bound */
    R.lb <- lb <- min{lb' : <p',R',I',lb'> received}
    R    <- E  <- R intersect I
                        /* E is used only in the proof */

    if I is maximal in N then
      if R is contained in either top(I) or bot(I)
         then chop()
         else split()
  until |I'| = 1 for all I' in N

  v <- I.lb     /* I = [v,v] */
  return(v);
end.
```

Figure 5: The increment register $\mathcal{I}$.

exchanges its interval and range with other processors, extends its range if possible, and splits or chops its interval and range whenever it finds that its interval is maximal among its competitors. When processor p's interval contains a single value, it continues broadcasting its interval and range until all competing intervals contain a single value, then p chooses its value and halts.

5.2.1 Correctness

Proving the correctness of this algorithm consists of proving two properties.

The first property we must prove is that given two nonoverlapping increments, the value returned by the first is less than the value returned by the second. This will imply

```
chop()
  begin
    I  <- smallest well-formed interval containing R
  end.

split()
  begin
    rank  <- rank of p in C  /* 0 is the lowest rank */
    value <- R.lb + rank
    if value in top(I) then
      I.lb <- R.lb <- I.lb + |I|/2
    else
      I.ub <- R.ub <- I.ub - |I|/2
    fi
  end.
```

Figure 6: Chopping and splitting an interval.

that the implementation is linearizable. In fact, this is very easy to prove using the observation that the ranges effectively isolate distinct generations, a fact mentioned earlier in the discussion of ranges:

Lemma 10: Suppose p and q are generation i and j processors returning values v and w, respectively. If $i < j$, then $v < w$.

Proof: Notice that v is no higher than p's upper bound. Notice also that p set its upper bound to $|IncSet| - 1$ at the end of phase 0 of generation i, and then broadcast this set to all processors in phase 1 of generation i. Since $i < j$, phase 1 of generation i is no later than phase 0 of generation j. Consequently, all generation j processors have received $IncSet$ before they set their lower bound at the end of phase 0 of generation j. This means that no generation j processor will ever lower its lower bound below p's upper bound. Since w is above q's lower bound, we have $v < w$. □

For the second property, remember that C is the set of competitors, and notice that E (a history variable used only in the proof) is the extended range (the result of dropping the lower bound of the real range R) that is used by a processor to assign values to its competitors (including itself). The second property we must prove is that $|C| \leq |E|$ for every processor p in every phase. This invariant says that p can always assign distinct values from E to its competitors. This will imply that the algorithm terminates: whenever a processor finds that its interval is maximal, it can assign itself a value and split or chop to a smaller interval containing this value. This will also imply that distinct processors choose distinct values: if p and q return the same value v, then at some point they both have the same extended range E consisting of the single value v and they both have a set of competitors C including p and q, but $|C| = 2 \not\leq 1 = |E|$.

```
initialize()
  begin
    k      <- current round number /* choose generation */
    p      <- <processor id, k> /* choose id */
    IncSet <- IncSet union {p} /* set of incrementors */
  end.

phase0()
  begin
    broadcast <p,IncSet>
    receive   <p',IncSet'> from all processors p'

    IncSet <- union of all IncSet' received
    GenSet <- set of all processors p' in IncSet with
                                           generation k' < k
    R.ub   <- |IncSet| - 1
    R.lb   <- lb <- |GenSet|
  end.

phase1()
  begin
    broadcast <p,R,lb>
    receive all <p',R',lb'>

    R.lb <- lb <- min gen k lower bound lb' received
    I    <- smallest well-formed interval containing R
  end.
```

Figure 7: The initialization phases.

Proving that $|C| \leq |E|$ requires reasoning about the interactions between the splits and chops performed by different processors in different phases, and we prove two claims (Claims 13 and 14 below) about these interactions. Let us fix a generation k for the rest of this section. We denote the values of I and R broadcast by p during phase r of an execution e by $I_{e,p,r}$ and $R_{e,p,r}$, and we denote the values of E and C held by p at the end of phase r of execution e by $E_{e,p,r}$ and $C_{e,p,r}$. We often omit subscripts like e and p when they are clear from context.

We say that *p splits to I in phase i* if p sends $\hat{I}$ in phase $i-1$ and I in phase i, where p changes from $\hat{I}$ to I by splitting. We say that p *splits up* or *splits down* depending on whether $I = top(\hat{I})$ or $I = bot(\hat{I})$. We say that *p chops into I in phase i* if p sends $\hat{J} \not\subseteq I$ in phase $i-1$ and $J \subseteq I$ in phase i, where p changes from $\hat{J}$ to J by chopping. We say that p *chops up* or *chops down* depending on whether $J \subseteq top(\hat{J})$ or $J \subseteq bot(\hat{J})$. Two simple properties about splitting and chopping are often useful.

Fact 11: If p splits from I_{i-1} to I_i, then the upper bounds of R_i, E_i and I_i, where $R_i \subseteq E_i \subseteq I_i$, are equal if p splits down, and the lower bounds are equal if p splits up.

Fact 12: If p chops from I_{i-1} to I_i, then the upper bounds of E_{i-1}, E_i, R_i, I_{i-1} and I_i, where $E_{i-1} = R_i \subseteq E_i \subseteq I_i \subset I_{i-1}$, are equal if p chops up, and the lower bounds are equal if p chops down.

The first property follows from the fact that the range spans the midpoint of the interval during a split (so the split truncates the range and interval at the same point). The second property follows from the fact that the initial range always spans the midpoint of the initial interval, so a split must occur before a chop (and again the split truncates the range and interval at the same point).

Reasoning about one processor p's splitting and chopping usually involves reasoning about another processor q's behavior in earlier phases. The first claim below argues that whenever a processor p with interval I has to find room for its competitors C in its extended range E, each of these competitors themselves had to find room for C in their extended ranges when they split or chopped into the interval I.

Claim 13: If $I_{q,j} \supseteq I_{p,i}$ for some $j < i$, then $C_{q,j} \supseteq C_{p,i}$.

Proof: Let r be a processor in $C_{p,i}$. This means that the interval $I_{r,i}$, sent by r to p in phase i, intersects $I_{p,i}$. Since $j < i$, the interval $I_{r,j}$, sent by r to q in phase j, contains $I_{r,i}$. Now, since $I_{q,j}$ contains $I_{p,i}$, and $I_{r,j}$ contains $I_{r,i}$, the fact that $I_{r,i}$ intersects $I_{p,i}$ implies that $I_{r,j}$ intersects $I_{q,j}$. Hence, $r \in C_{q,j}$. It follows that $C_{p,i} \subseteq C_{q,j}$. $\square$

The second claim we prove concerns the fact that a processor p may split to an interval I in an orderly sequence of splits while another processor q may chop into I in a chaotic interleaving of splits and chops. The claim states that the moment this happens, p's extended range E spans its entire interval I from that moment on. This means that if chopping complicates our analysis in one way, it simplifies our analysis in another since we no longer have to be careful to distinguish between intervals and ranges.

Claim 14: Suppose p splits to I in phase i, and suppose q chops into I in phase j. If $i \leq \ell$ and $j \leq \ell$, then $I_{p,\ell} = E_{p,\ell}$ at the end of phase ℓ.

Proof: Suppose p splits down from $\hat{I}$ to I, and let E be p's extended range at the end of phase i. Since p split down, we know that $I.ub = E.ub$ by Fact 11. Since the upper bounds of $I_{p,\ell}$ and $E_{p,\ell}$ are still equal at the end of phase ℓ, all we have left to show is that their lower bounds are equal. First, notice that q must have chopped down and not up: this follows from the fact that p split down from $\hat{I}$ to I and the fact that q chopped from $\hat{J} \not\subseteq I$ to $J \subseteq I$. Let R be the range q sent together with J in phase j. According to Fact 12, the fact that q chopped down from $\hat{J}$ to J implies that $R.lb = J.lb$. Furthermore, the fact that q chopped down from $\hat{J} \not\subseteq I$ to $J \subseteq I$ implies that $J.lb = I.lb$. It follows that $R.lb = J.lb = I.lb$. On the other hand, since $R.lb = I.lb \leq I_{p,\ell}.lb$ and since p drops its lower bound every round, it follows that $I_{p,\ell}.lb = E_{p,\ell}.lb$ by the end of phase $\ell \geq j$.

Suppose p split up from $\hat{I}$ to I, and let E be p's extended range at the end of phase i. Since p split up, we know that $I.lb = E.lb$ by Fact 11, and we will now show that $I.ub = E.ub$ as well. It will follow that $I = E$ at the end of phase i, and hence that $I_{p,\ell} = E_{p,\ell}$ at the end of phase $\ell \geq i$. Suppose on the contrary that $E.ub < I.ub$. This means that the upper bound $R_{p,2}.ub$ of p's phase 2 range is also less than $I.ub$. It follows that $R_{p,2}.lb < I.lb$, since otherwise $R_{p,2} \subseteq I$ and p would have chosen I as its initial interval—the smallest well-formed interval containing $R_{p,2}$—and not an interval as large as $\hat{I}$. On the other hand, since q's initial interval $I_{q,2}$ contains q's interval $\hat{J}$, and since $\hat{J}.lb$ is under $I.lb$, we know that $I_{q,2}.lb \leq \hat{J}.lb < I.lb$. Thus, $R_{p,2}.lb < I.lb$ and $I_{q,2}.lb < I.lb$, so we know that q will drop its lower bound below $I.lb$ at the end of phase 2, and that q's lower bound will remain below $I.lb$ until q splits up to an interval with a lower bound at or above $I.lb$. However, since $\hat{J}.lb < I.lb$, we know that q's lower bound is still below $I.lb$ at the end of phase $j-1$, so it is impossible for q to have chopped up from an interval $\hat{J}$ containing I in phase $j-1$ to an interval J contained in I in phase j, a contradiction. $\square$

These two claims give us the tools we need to prove that $|C| \leq |E|$ is an invariant. We prove this invariant by defining the condition

$\mathcal{I}_\ell$: $|C_{e,p,r}| \leq |E_{e,p,r}|$ in all executions e for all processors p and generation k phases $r = 2, \ldots, \ell$,

and then proceeding by induction on $\ell \geq 2$ to prove that $\mathcal{I}_\ell$ holds for all ℓ. Fix some execution e and processor p, and let I, R, E, and C denote $I_{e,p,\ell}$, $R_{e,p,\ell}$, $E_{e,p,\ell}$, and $C_{e,p,\ell}$.

As the basis of our induction, we show that the invariant is true initially. We actually prove two results. The first concerns the simple case where p's range contains some other process's initial range, and the second concerns the more common case where p's interval (which is bigger than the range) contains some other process's initial interval.

Claim 15: If R contains some processor q's initial range $R_{q,1}$, then $|C| \leq |E|$.

Proof: Let r be a processor in C. This means that r sent an interval intersecting I to p in phase ℓ, and therefore that r sent a message to q in phase 0. Since the size of $R_{q,1}$ is exactly the number of processors sending to q in phase 0, it follows that $|C| \leq |R_{q,1}| \leq |R|$. Finally, $|R| \leq |E|$ since $R \subseteq E$. $\square$

Claim 16: If I contains some processor q's initial interval $I_{q,2}$, then $|C| \leq |E|$.

Proof: We will prove that either $R_{p,1} \subseteq R$ or $R_{q,1} \subseteq R$, depending on $R_{p,1}$'s upper bound, and in either case we will be done by Claim 15.

Suppose $R_{p,1}.ub < I.lb$. This case can never arise, since the upper bound of p's range never increases, and since p never splits or chops to an interval above its upper bound.

Suppose $R_{p,1}.ub \in I$. If we also have $R_{p,1}.lb \in I$, then $R_{p,1} \subseteq I$ which implies $R_{p,1} \subseteq R$ and we are done, so suppose that $R_{p,1}.lb < I.lb$. In this case, we know that $R_{q,2}.lb \leq R_{p,1}.lb < I.lb$ since q lowers its lower bound to $R_{p,1}.lb$ or lower at the end of phase 1 before choosing its new range and interval for phase 2. This means that $R_{q,2} \not\subseteq I$, but this in turn leads to the contradiction $I_{q,2} \not\subseteq I$ since $R_{q,2} \subseteq I_{q,2}$, so this case can never arise.

Suppose $R_{p,1}.ub > I.ub$. Since the upper bound of p's initial range is above I, we know that p has split down at least once, and hence that $R.ub = I.ub$ by Fact 11. Furthermore, since p lowered its lower bound to $R_{q,1}.lb$ or lower at the end of phase 1 before choosing its new range and interval for phase 2, we know that p's lower bound will remain $R_{q,1}.lb$ or lower until p splits up to an interval with a lower bound above $R_{q,1}.lb$. However, since $R_{q,1} \subseteq R_{q,2} \subseteq I_{q,2} \subseteq I$, we have

$$R.ub = I.ub \geq R_{q,1}.ub \geq R_{q,1}.lb \geq R.lb,$$

so $R_{q,1} \subseteq R$. □

As for the inductive step itself, if I does not contain the initial interval of any processor, then all of p's competitors have split or chopped into I. The next result concerns the chopping case. It says that if I is p's interval and if any processor q has chopped into I at any time in the past—regardless of whether p and q are now competitors—then the invariant is preserved. It is a strong statement that chopping quickly brings distinct intervals and ranges into synch.

Claim 17: Suppose $\mathcal{I}_{\ell-1}$ is true. If any processor has chopped into I by phase ℓ, then $|C| \leq |E|$.

Proof: Suppose I is p's initial interval, or contains any other process's initial interval. Then we are done by Claim 16.

Suppose p itself chops from $\hat{I}$ to I in phase $i \leq \ell$. Since p chopped its interval from $\hat{I}$ to I in phase i, it follows that $C \subseteq C_{p,i-1}$ by Claim 13. Since p does not split its interval in phases i through ℓ, we know that $E_{p,i-1} \subseteq E$. Consequently, since $i-1 \leq \ell-1$, it follows from $\mathcal{I}_{\ell-1}$ that $|C| \leq |C_{p,i-1}| \leq |E_{p,i-1}| \leq |E|$, as desired.

Suppose p splits from $\hat{I}$ to I, and that some other processor q chops from $\hat{J} \not\subseteq I$ to $J \subseteq I$ in some phase $j \leq \ell$. Since q chopped from $\hat{J}$ to J, it follows that $C \subseteq C_{q,j-1}$ by Claim 13. In addition, since q chopped from $\hat{J}$ to J, we know that $E_{q,j-1} \subseteq J \subseteq I$. Finally, we know that $I = E$ by Claim 14 since $j-1 \leq \ell-1$. It therefore follows from $\mathcal{I}_{\ell-1}$ that $|C| \leq |C_{q,j-1}| \leq |E_{q,j-1}| \leq |E|$, as desired. □

The difficult cases, therefore, are the cases in which p and all its competitors split from $\hat{I}$ to I. The case of splitting down is easy, but the case of splitting up is difficult. In fact, understanding how to choose and manipulate ranges to make the case of splitting up go through is the most important way in which the increment register algorithm differs from the strong renaming algorithm it is based on.

Claim 18: Suppose $\mathcal{I}_{\ell-1}$ is true. If p and all its competitors have split down to I by phase ℓ, then $|C| \leq |E|$.

Proof: Let q be the greatest competitor in C. This means that q is the greatest processor to send an interval contained in I to p in phase ℓ. Consider the phase j in which q split from $\hat{I}$ to I, and notice that $C \subseteq C_{q,j-1}$ by Claim 13. Since q is the greatest processor in C and since q split down from $\hat{I}$ to I, processor q found that all processors in $C \subseteq C_{q,j-1}$ could choose distinct values from the bottom half of its extended range $E_{q,j-1}$, where the bottom half of its extended range just happens to be $R_{q,j} = I \cap E_{q,j-1}$. Consequently, $|C| \leq |R_{q,j}|$. We will now show that $R_{q,j} \subseteq E$, and it will follow that $|C| \leq |E|$, as desired. First, notice that $R_{q,j} \subseteq I_{q,j} = I$. Next, notice that $I.ub = E.ub$ by Fact 11 since p split down, so $R_{q,j}.ub \leq I.ub = E.ub$. Finally, notice that $j \leq \ell$ and that p lowers its lower bound as much as possible every phase, so $E.lb \leq R_{q,j}.lb$ by the end of phase ℓ. It follows that $R_{q,j} \subseteq E$ as desired. $\square$

Finally, let us consider the tricky case of splitting up.

Claim 19: Suppose $\mathcal{I}_{\ell-1}$ is true. If p and all its competitors have split up to I by phase ℓ, then $|C| \leq |E|$.

Proof: Let q be the least competitor in C. This means that q is the least processor to send an interval contained in I to p in phase ℓ. Consider the phases $i \leq \ell$ and $j \leq \ell$ in which p and q split up from $\hat{I}$ to I, respectively. Notice that since p and q split their intervals at the ends of phases $i-1$ and $j-1$, Claim 13 implies that $C \subseteq C_{p,i-1}$ and $C \subseteq C_{q,j-1}$.

Suppose that $i \leq j$ (the case with $j \leq i$ is similar, and easier). Let e' be the execution differing from e only in that in each phase $k \geq i-1$ of e' the processors p and q receive messages from exactly the same set of processors that p receives messages from in the corresponding phase of e. Notice that this does not change the set of messages p receives in phase $i-1$, and hence does not change the fact that p splits up to I in phase i, but it might change the messages and splitting of q.

First, consider the lower bounds of the extended ranges that p and q use when they decide to split up at the end of phases $i-1$ and $j-1$ in e. At the end of phase $i-1$, processor p first computes the lower bound $lb_{e,p,i-1}$ and then uses this lower bound to set the lower bound of its extended range $E_{e,p,i-1}$ to the maximum of $lb_{e,p,i-1}$ and the lower bound of $\hat{I}$. It then broadcasts $lb_{e,p,i-1}$ to q in phase $i \leq j-1$. Consequently, at the end of phase $j-1$, processor q sets $lb_{e,q,j-1}$ to $lb_{e,p,i-1}$ or lower, and uses this lower bound to set the lower bound of its extended range $E_{e,q,j-1}$ to the maximum of $lb_{e,q,j-1}$ and the lower bound of $\hat{I}$. In other words, $E_{e,p,i-1}.lb \geq E_{e,q,j-1}.lb$. In fact, the construction of e' from e guarantees that $E_{e',q,i-1}.lb = E_{e',p,i-1}.lb = E_{e,p,i-1}.lb \geq E_{e,q,j-1}.lb$.

Next, consider the set of competitors for p and q in e. It follows from Claim 13 that $C_{e,q,j-1} \subseteq C_{e,p,i-1}$. In fact, from the construction of e' from e, it follows that $C \subseteq C_{e,q,j-1} \subseteq C_{e,p,i-1} = C_{e',p,i-1} = C_{e',q,i-1}$.

The conclusion of this little exercise is that at the end of phase $i-1$ in e' processor q's lower bound is as high and its set of competitors is as large as at the end of phase $j-1$ in e. Since q splits up at the end of phase $j-1$ in e, it will split up at the end of phase $i-1$ in e', assuming its interval $\hat{I}$ is maximal among the intervals it receives in e'. It must be maximal, however, because q receives precisely the same intervals in phase $i-1$ of e' as p does, and p splits up. In fact, p and q assign the same values to the same processors at the end of phase $i-1$ of e'. It follows from $\mathcal{I}_{\ell-1}$ that p and q can assign distinct values from $E_{e',p,i-1}$ and $E_{e',q,i-1}$ to all processors in $C_{e',p,i-1} = C_{e',q,i-1}$, and we have already argued that they do so in precisely the same way. Since q is the smallest processor in $C \subseteq C_{e',q,i-1}$ and q splits up, this means that both p and q can find values for all processors in C in the top halves of their extended ranges. Since the top half of p's extended range is E—remember that upper bounds never change—it follows that $|C| \leq |E|$, as desired. □

Putting all of this together, we have our invariant:

Lemma 20: $|C| \leq |E|$.

Proof: We proceed by induction on $\ell \geq 2$ to prove that $\mathcal{I}_\ell$ is true for all ℓ.

First suppose $\ell = 2$. Since $I_{p,2}$ itself is an initial interval contained in $I_{p,2}$, it follows by Claim 16 that $|C_{p,2}| \leq |E_{p,2}|$.

Now suppose $\ell > 2$ and $\mathcal{I}_{\ell-1}$ is true. If I contains some process's initial interval, then we are done by Claim 16. If some processor chops into I, then we are done by Claim 17. If all processors split from $\hat{I}$ to I, then we are done by Claims 18 and 19. □

Using this invariant and Lemma 10 we can prove that our implementation is correct:

Theorem 21: $\mathcal{I}$ is a linearizable, wait-free implementation of an increment register.

Proof: First, notice that all nonfaulty processors choose a value. This follows from the fact that, given any phase i of any generation k, all generation k intervals of maximal size in phase i will either split or chop at the end of phase i or $i + 1$, meaning that the size of the maximal generation k interval decreases by a factor of at least two with every two rounds. Thus, eventually all generation k processors will have intervals of size 1 and choose a value.

Second, notice that two processors always return distinct values. If p and q are of distinct generations, then the result follows by Lemma 10. If p and q are of the same generation and both return the same value v, then they both have the same extended range E consisting of the single value v and they both have the same set of competitors C consisting of p and q, but $|C| = 2 \not\leq 1 = |E|$, violating the invariant $|C| \leq |E|$.

Third, we need to show that if p chooses the value v, then there are at least $v - 1$ other processors of p's generation or earlier, which, if they decide, decide on values below v. Consider the highest upper bound U chosen by any processor q in p's generation. There are at least U processors of p's generation or earlier which, if they decide, decide on values less than or equal to U. Therefore there are at least $v - 1$ of these processors which decide on values $w < v$ if they decide at all.

Finally, it follows from Lemma 10 that the algorithm is linearizable. □

5.2.2 Time complexity

We now show that increment operations halt in $O(\log c)$ rounds, where c is the number of concurrent operations. Technically speaking, a failed operation is concurrent with (or overlaps) every following operation, so c can grow artificially large. Fortunately, we can prove a tighter bound, depending on a set of concurrent operations that is generally a much smaller set.[1] Our algorithm has the nice property that the invocation

[1]This does not mean that our algorithm runs faster than the $\Omega(\log c)$ worst-case lower bound, because these two sets are equal in that single worst-case execution.

of an increment operation delays at most one generation. If the invoking processor is nonfaulty, then the increment delays its own generation. If the invoking processor is faulty, then it may delay a later generation, but it will delay at most one. In fact, we can identify exactly which generation an operation delays.

For each generation k, we define the *active* set of processors, namely those processors or invocations that contribute to the generation's running time. We show that the largest range chosen by any generation k processor is bounded in size by the size of the active set, and we show that a generation halts in time logarithmic in the size of the largest range. From this it follows that all generation k increment operations halt in time $\log c_k$, where c_k is the size of the active set for generation k.

Active Sets We begin by defining $active_k$, the *active set* of processors for generation k.

Loosely speaking, the active set for generation k consists of all processors that the "good" processors learn about for the first time in round k. Remember that all processors choose their initial range at the end of phase 0, exchange their ranges, and then choose their initial intervals at the end of phase 1 based on the ranges they receive. The "good" processors for generation k are the generation k processors that survive these initialization phases and begin broadcasting intervals.

Let gen_k be the set of generation k processors. Formally, we define $good_k$ to be the set of generation k processors that are nonfaulty in phases 0 and 1 of generation k (that is, they do not fail in rounds k and $k+1$). For any good processor p, the set of processors that p has learned about in the first k rounds is exactly the value of its set $IncSet$ at the end of round k, which we denote by $IncSet_{p,k}$. The set $known_k$ of all processors the good processors know about at the end of round k is given by

$$known_k = \bigcup_{p \in good_k} IncSet_{p,k},$$

and the set $active_k$ of all processors that the good processors learn about for the first time in round k is

$$active_k = known_k - known_{k-1}$$

(where "−" denotes set difference).

It is clear that the set of known processors is nondecreasing:

Claim 22: $known_{k-1} \subseteq known_k$ for all k.

Proof: If $q \in known_{k-1}$, then $q \in IncSet_{p,k-1}$ for some $p \in good_{k-1}$. This means that p survived phase 1 of generation $k-1$ and successfully broadcast its set $IncSet_{p,k-1}$ to all processors in that phase. Since phase 1 of generation $k-1$ is phase 0 of generation k, this means that $IncSet_{p,k-1} \subseteq IncSet_{r,k}$ at the end of phase 0 of generation k for all good processors $r \in good_k$ in generation k. Thus, $q \in IncSet_{p,k-1} \subseteq known_k$, and it follows that $known_{k-1} \subseteq known_k$. □

Using this observation, we can show that the set $active_k$ has two desirable properties: every nonfaulty generation k processor belongs to $active_k$, and every processor belongs to at most one set $active_k$.

Claim 23: $good_k \subseteq active_k$ for all k, and $active_j \cap active_k = \emptyset$ for all $j \neq k$.

Proof: First, notice that if $p \in good_k$ then p is a generation k processor that survives phase 0 of generation k and adds p to its own set $IncSet_{p,k}$. Notice also that a generation k processor cannot appear in any set $IncSet_{q,j}$ for any generation j processor q, where $j < k$. It follows that $p \in good_k$ implies $p \in known_k - known_{k-1} = active_k$.

Next, the remainder of the claim follows immediately from the fact that $known_j \subseteq known_{k-1}$ for all $j \leq k-1$, which follows from Claim 22. □

Maximal Range For each generation k, we can bound the size of the ranges sent by good processors with $active_k$. Since we are trying to bound the execution time of generation k increments, we need only consider the ranges of the good processors, since all other processors fail by the end of phase 1.

Consider the largest range a good processor p can send. Every processor p chooses upper and lower bounds u_p and l_p at the end of phase 0, and then never raises its lower bound without splitting or chopping up to a smaller interval and range. At any given time, a processor p's lower bound is the minimum of the lower bounds l_q chosen by some subset of the generation k processors. In the worst case, a good processor p's largest range $R_{p,i}$ is contained in $max\_range_k = [lb_k, ub_k]$, where

$$\begin{aligned} ub_k &= \max\{u_p : p \in good_k\} \\ lb_k &= \min\{l_p : p \in gen_k\} \end{aligned}$$

In other words,

Claim 24: $R_{p,i} \subseteq max\_range_k$ for every good processor $p \in good_k$ and every phase i.

The next result shows that the size of max_range_k is bounded by the size of $active_k$, and hence so is the size of any range used by any good processor in generation k.

Claim 25: $|max\_range_k| \leq |active_k|$.

Proof: We prove that $|known_k| \geq ub_k + 1$ and that $|known_{k-1}| \leq lb_k$, and it follows that

$$\begin{aligned} |max\_range_k| &= ub_k - lb_k + 1 \\ &\leq |known_k| - |known_{k-1}| = |known_k - known_{k-1}| = |active_k| \end{aligned}$$

since $known_{k-1} \subseteq known_k$ by Claim 22.

To prove $|known_k| \geq ub_k + 1$, consider the good processor $p \in good_k$ with the maximum upper bound $u_p = ub_k$ at the end of phase 0. Processor p chose u_p to be $|IncSet_{p,k}| - 1$ and $IncSet_{p,k} \subseteq known_k$, so $|known_k| \geq u_p + 1$.

To prove $|known_{k-1}| \leq lb_k$, consider the processor $p \in gen_k$ with the minimum lower bound $l_p = lb_k$ at the end of phase 0. Since all good processors $g \in good_{k-1}$ survive phases 0 and 1 of generation $k-1$, they all send their sets $IncSet_{g,k-1}$ to p during round k (that is, during phase 1 of generation $k-1$), so p has heard of all processors in $known_{k-1}$ before it sets its lower bound l_p at the end of round k (that is, during phase 0 of generation k). Consequently, $|known_{k-1}| \leq lb_k$. □

Running Time Analysis For each generation k, we can bound the size of intervals sent by good processors with *max_range*$_k$. Consider any telescoping chain

$$I_1 \supset I_2 \supset \cdots \supset I_l$$

of intervals sent during phase 2, where I_i strictly contains I_{i+1}, and suppose the sequence is of maximal length. Since I_1 is maximal, we know that it will split in half immediately at the end of phase 2, leaving $I_2 \supset \ldots \supset I_l$ as a maximal chain. We now prove that the size of I_2 is roughly the size of *max_range*$_k$. Since the size of the maximal interval reduces by half in each round, the running time is clearly logarithmic in the size of the largest interval, and it will follow that the running time is roughly logarithmic in $|\mathit{max\_range}_k| \leq |\mathit{active}_k|$.

Claim 26: Given any sequence of intervals $I_1 \supset I_2 \supset \cdots \supset I_l$ sent in phase 2 of generation k, we have $|I_2| \leq 2|\mathit{max\_range}_k|$.

Proof: Since the intervals I_i in the chain are sent in phase 2, they are sent by good processors in *good*$_k$ (processors surviving phases 0 and 1), and their ranges R_i are contained in *max_range*$_k$ by Claim 24. The upper and lower bounds $R_1.ub$ and $R_1.lb$ of R_1 are clearly in the top half and bottom half of I_1, respectively. Since I_2 is strictly contained in I_1, we know that I_2 is either in the top or bottom half of I_1. We consider the two cases separately.

Suppose I_2 is in the top half of I_1. Then since the lower bound $R_1.lb$ of R_1 at the end of phase 1 is in the bottom half of I_1, the lower bound $R_2.lb$ of R_2 will drop to the bottom of I_2 at the end of phase 2. Since the upper bound $R_2.ub$ of R_2 is in the top half of I_2, the range R_2 will span the bottom half of I_2 by the end of phase 2. This means that $|I_2| \leq 2|R_2| \leq 2|\mathit{max\_range}_k|$.

Suppose I_2 is in the bottom half of I_1. This means that at the end of phase 1, the lower bound $R_2.lb$ of R_2 is in the bottom half of I_2. At the end of phase 2, therefore, the lower bound $R_1.lb$ of R_1 will be in the bottom half of I_2. Since the upper bound $R_1.ub$ of R_1 is in the top half of I_1, it follows that R_1 will span the top half of I_2:

$$|I_2| \leq 2|R_1| \leq 2|\mathit{max\_range}_k|.$$

$\square$

Combining these results, we are done:

Theorem 27: Every generation k increment operation completes within $O(\log |\mathit{active}_k|)$ rounds.

Proof: By Claim 26, after phase 2 starts, for every telescoping chain of intervals, the set of active processors is guaranteed to be at least half of the size of the second interval in the chain, so as soon as the first interval splits and chops (which will happen immediately since it is immediately maximal), the chain will disappear in time logarithmic in the number of active processors. $\square$

6 Conclusion

This paper represents an additional step toward understanding the round complexity of problems in synchronous message-passing systems. We observed that as long as processors remain in order-equivalent states, they cannot solve any problem that requires processors to take distinct actions. We then showed that any comparison-based protocol has an execution in which order-equivalence is preserved for $\log c$ rounds with c participating processors, and that this bound is tight. This lower bound on order-inequivalence yields the best-known lower bounds for a variety of concurrent objects, including increment registers, ordered sets, and related data types, as well as for decision tasks such as strong renaming.

We have also seen that this logarithmic bound separates protocols that can and cannot solve nontrivial problems: we have seen examples of two nontrivial problems, the strong renaming task and increment register objects, that have solutions with complexity lying at exactly this boundary. These implementations are substantially more efficient than an $O(n)$ general-purpose algorithm using consensus, especially since the degree of concurrency c itself is typically much less than n, the total number of processors.

A second interesting aspect of our construction is that our optimal increment register implementation is based on our optimal solution to strong renaming, although these two problems might seem quite different at first glance. Concurrent object implementations are usually more difficult than solutions to decision tasks. Unlike decision tasks, where processors start simultaneously, compute for a while, and halt with their outputs, concurrent objects have unbounded lifetimes during which they must handle an arbitrary number of operations, these operations can be invoked at any time, and the order in which operations are invoked is often important.

We can now draw a more complete picture of the complexity hierarchy for this model. We have shown here that a logarithmic number of rounds is the minimal necessary to solve nontrivial problems. It is known that a linear number of rounds is the most needed to solve such problems (by reduction to consensus). In between, it is known that the k-set agreement task [CHLT93] requires $\lfloor n/k \rfloor + 1$ rounds, well above the logarithmic lower bound for strong renaming, but less than the $n + 1$ bound for consensus. Little is known about other sublinear problems in this model.

Finally, we observe our lower bound on order-equivalence in the synchronous model translates into a similar bound in the semi-synchronous model as well. In this model, processors take steps at a rate bounded from above and below by constants, and message delivery times vary between 0 and d. Our lower bound for order-inequivalence translates into an immediate $\Omega(d \log c)$ lower bound in the semi-synchronous model. It would be interesting to see if that bound could be improved.

Acknowledgments

Early versions of these results were originally published in [HT90] and [CT94]. We thank three anonymous referees for their helpful and numerous comments on earlier drafts of this paper. The first author was supported in part by NSF grant CCR-93-08103.

References

[ABND+87] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Daphne Koller, David Peleg, and Rudiger Reischuk. Achievable cases in an asynchronous environment. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, pages 337–346, October 1987.

[ABND+90] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rudiger Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, July 1990.

[CHLT93] Soma Chaudhuri, Maurice Herlihy, Nancy Lynch, and Mark R. Tuttle. A tight lower bound for k-set agreement. In *Proceedings of the 34th IEEE Symposium on Foundations of Computer Science*, pages 206–215. IEEE, November 1993.

[CT94] Soma Chaudhuri and Mark R. Tuttle. Fast increment registers. In Gerard Tel and Paul Vitányi, editors, *Proceedings of the 8th International Workshop on Distributed Algorithms*, volume 857 of *Lecture Notes in Computer Science*, pages 74–88. Springer-Verlag, Berlin, October 1994.

[FL82] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, June 1982.

[FL87] G.N. Frederickson and N.A. Lynch. Electing a leader in a synchronous ring. *Journal of the ACM*, 34(1):98–115, January 1987.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[Her91a] Maurice Herlihy. Randomized wait-free concurrent objects. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 11–22. ACM, August 1991.

[Her91b] Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HF89] Joesph Y. Halpern and Ronald Fagin. Modelling knowledge and action in distributed systems. *Distributed Computing*, 3(4):159–179, 1989.

[HM90] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, July 1990.

[HS93] Maurice P. Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 111–120. ACM, May 1993.

[HT90] Maurice P. Herlihy and Mark R. Tuttle. Wait-free computation in message-passing systems: Preliminary report. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 347–362. ACM, August 1990.

[HW90] Maurice P. Herilhy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.

[Lam89] Leslie Lamport. The part-time parliament. Technical Report 49, DEC Systems Research Center, September 1989.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[MT88] Yoram Moses and Mark R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3(1):121–169, 1988.

[PSL80] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[Sch87] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. Technical report, Cornell University, Computer Science Department, November 1987.

[SP89] E. Styer and G.L. Peterson. Tight bounds for shared memory symetric mutual exclusion problems. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 177–192, August 1989.

Wait-Free Implementations in Message-Passing Systems

Soma Chaudhuri Maurice Herlihy
Mark R. Tuttle

CRL 98/5
May 1998