Fast Increment Registers

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

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

Abstract

We give an optimal, wait-free implementation of an *increment register*. An increment register is a concurrent object consisting of an integer-valued register with an *increment* operation that atomically increments the register and returns the previous value. We implement this register in a synchronous, message-passing model with crash failures. In our implementation, an increment operation halts in $O(\log c)$ rounds of communication, where c is the number of concurrently executing increment operations. This is the first wait-free implementation of any object that matches the $\Omega(\log c)$ lower bound by Herlihy and Tuttle for wait-free implementations, and it proves that their lower bound is tight. The significance of our result is not so much the implementation itself, but what it says about lower bounds. Our result says that $\Omega(\log c)$ is the best possible lower bound that applies to so many objects in so many models. The algorithm itself is interesting, however, because it is based on an optimal solution for *strong renaming*, a simple decision problem used by Herlihy and Tuttle to prove their lower bound.

1 Introduction

A *concurrent object* is a data structure that can be accessed by many processes simultaneously. Most interesting implementations of concurrent objects are designed for asynchronous systems of unreliable processes. Most of these implementations depend on some form of mutual exclusion—involving locks or semaphores—to restrict access to the object. In such implementations, a process must be inside the critical section before it can perform an operation on the object, and this guarantees that the process can access and modify the object in isolation without interference from other processes. Unfortunately, implementations based on mutual exclusion can be unacceptable in asynchronous, unreliable systems, since a process (possibly holding a lock) can fail in the critical section and block other processes from accessing the object. Even when processes do not fail, processes may be delayed in the critical section (due to a page fault or being swapped out as the result of the expiration of a scheduling quantum), and again block other processes from accessing the object. In addition, if processes run at different speeds, then a fast process can be blocked by a slow process as it plods through the critical section.

In contrast, an implementation of an object is said to be *wait-free* if it guarantees that any nonfaulty process can complete any operation on the object in a finite number of steps, independent of the failure of other processes or variations in their speed. Wait-free implementations provide a strong form of concurrency and fault-tolerance since they guarantee that no process can be prevented from completing an operation by the failure of other processes, or by differences in their speeds. They also provide a kind of real-time

*Supported in part by NSF grant CCR-93-08103.

guarantee since they guarantee a bound on the number of steps a process must take to complete an operation.

Concurrent objects are an important part of concurrent algorithms, so it is important to understand how quickly these objects can be implemented. With this goal in mind, Herlihy and Tuttle [HT90] prove a general-purpose lower bound for wait-free implementations of concurrent objects. They consider a synchronous, message-passing model with crash failures, and they prove that any wait-free implementation of any object that can solve *strong renaming* must have an operation requiring $\Omega(\log c)$ rounds of communication in the worst case.[1] Closer examination of their proof technique, however, reveals that their lower bound holds for a much larger set of objects, including any object with an operation that must return distinct values on distinct invocations. Notice that since there is no such thing as a slow process in a synchronous model, the notion of a wait-free implementation in this model coincides with the simpler notion of an implementation that can tolerate the failure of all but one process. However, since they prove their lower bound in this restrictive synchronous model, their lower bound is a general result that applies to any more asynchronous model in which slow processes do exist.

In this work, we prove that their lower bound is tight. We give an optimal, wait-free implementation of an *increment register*. An increment register is a concurrent object consisting of an integer-valued register with an *increment* operation that atomically increments the register and returns the previous value. This is a special case of a *fetch&add* register since 1 is the only value that can be added to the register. We implement this register in a synchronous, message-passing system with crash failures, the same model used to prove the lower bound. In our implementation, an increment operation halts in $O(\log c)$ rounds of communication, where c is the number of concurrently executing increment operations. This is the first wait-free implementation of any concurrent object to match the $\Omega(\log c)$ lower bound.

The primary significance of our work is what it has to say about proving lower bounds. The fact that we implement our increment register in the same powerful, synchronous model that Herlihy and Tuttle use to prove their lower bound says that their proof technique cannot be pushed any farther. In particular, it is not possible to prove a better general-purpose lower bound that applies to as many objects and to as many models of computation. It is likely that implementing a particular object in a particular model will require more than $O(\log c)$ steps. To prove this, however, will require considering a smaller class of objects (not the class of all objects) or a closer approximation of the model of interest (not the synchronous model).

Our implementation is also interesting on purely algorithmic grounds, since our optimal implementation of an increment register is based on an optimal solution to the strong renaming problem [HT90]. We find it remarkable that both the upper and lower bound for increment registers arise from considering the same decision problem. In general, implementing long-lived objects is inherently more difficult than solving decision problems. In the first place, a decision problem is solved once whereas the same operation can be invoked on an object repeatedly. In the second place, processes solving a decision problem start together simultaneously at time 0, whereas processes invoking operations on an object can arrive at different and unpredictable times. The major technical difficulty in this work has been to guarantee that processes invoking increment operations on the register at different times do not interfere with each other.

[1] Strong renaming is a decision problem in which processes begin with process ids taken from a totally-ordered set, and choose new names for themselves. The problem requires that if c processes participate in the protocol then these processes end up with distinct names in the range $1, \ldots, c$. This is a strong form of the general renaming problem [ABND+87, ABND+90] since the range of names chosen must equal the number of participants, which is known to be impossible in asynchronous systems.

Of course, there are many ways to implement an increment register. In fact, there are general-purpose techniques for constructing a wait-free implementation of any concurrent object. They are based on atomic broadcast [Lam78, Lam89, Sch87] and consensus [Her91b], so they yield implementations requiring $O(n)$ rounds where n is the number of processes. On the other hand, it is well-known that type-specific techniques often yield more efficient implementations than general-purpose techniques [Her86], but our implementation shows that this in complexity can be substantially greater than previously known. Prior to this, a $O(\sqrt{c})$ round implementation of an increment register [HT90] was the closest that any wait-free implementation of any object had come to meeting the $\Omega(\log c)$ lower bound.

Finally we note that an active area of research concerns asynchronous, wait-free data structures called *counting networks* [AHS91], and that counting networks yield easy implementations of increment registers. A counting network resembles a sorting network, except that the comparators in the sorting network are replaced with constructs called *balancers*. Counting networks can be used to construct fast increment registers, but counting networks are designed for asynchronous systems in which processes do not fail, and in the absence of failures an increment register can be implemented with a single round of communication in our synchronous model. On the other hand, in the presence of failures, comparing execution times is difficult since simulating the balancers seems to require attaining some degree of consensus, resulting in increment registers requiring a linear number of rounds rather than $O(\log c)$ rounds as required by our implementation.

The rest of this paper is organized as follows. In Section 2 we define our model of computation, and in Section 3 we define concurrent objects and their implementation. In Section 4, we give our optimal wait-free implementation of an increment register. The correctness of our algorithm is proven in Section 5, and its running time is analyzed in Section 6.

2 Model

Our model of computation is a standard synchronous, message-passing model with crash failures. We sketch the model here, and refer the reader to other papers [MT88, HM90, HF89] for details. A system consists of n unreliable *processes* $p_1, \ldots, p_n$ and an external *environment* p_0. We refer to $p_1, \ldots, p_n$ as system processes, and to p_0 as the environment process, which we consider identical to the system processes with three exceptions noted below. We assume that all processes share a *global clock*, which 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. Every round, every process sends messages to other processes, then receives the messages sent to it in that round, and then performs some local computation. We assume that any process can send a message to any other process. Communication is reliable, in that a message sent in one round is guaranteed to be delivered in the same round. Processes, however, may crash at any time, possibly in the middle of sending messages. The environment is not allowed to crash (exception number one).

A *global state* is a tuple $(s_0, s_1, \ldots, s_n)$ of local states, one local state s_i for each process p_i. A *local state* for process p_i contains its id, the time on the global clock, and the entire history of messages it has sent and received so far. In addition, the local state of the environment contains the failure information and any other information of relevance to the system that cannot be deduced from processes' local states (exception number two).

A *message sequence* M is a sequence $m_{00}, m_{01}, \ldots, m_{0n}, m_{10}, \ldots, m_{nn}$ of messages describing one round of communication. The interpretation of M is that m_{ij} is the message sent by p_i to p_j during the round, or $\perp$ if no message was sent. An *execution* e is an infinite sequence $g_0 M_1 g_1 M_2 \ldots$ of alternating global states and message sequences, where g_i is the

global state at time i and M_i is the sequence of messages sent in round i.

We assume that every system process is following a deterministic *protocol* that determines what actions it performs and what messages it sends. A process follows its protocol in every round, except that a process 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 process is considered *faulty* in an execution if it fails in some round of that execution, and *nonfaulty* otherwise. The environment process need not follow a protocol (exception number three).

3 Concurrent Objects

An *object* is a data structure that can be accessed concurrently by all processes. 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 process 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 just a set of sequential histories describing the sequential behavior of the object.

As an example, an *increment register* is just a register with an *increment* operation. The value of the register is an integer, initially 0. 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 values in the order $0, 1, 2, \ldots$.

We are interested in concurrent implementations of such objects. To us, given an object O intended to be used by n processes $P_1, \ldots, P_n$, an implementation of O will be a collection of n processes $F_1, \ldots, F_n$ called *front ends* [Her91a] that process the invocations from the $P_1, \ldots, P_n$ and return the responses from O. In our model, we assume that the system processes $p_1, \ldots, p_n$ are really the front ends $F_1, \ldots, F_n$. We assume that the invoking processes $P_1, \ldots, P_n$ are part of the environment process p_0, and we ignore them completely. With this in mind, we define a *history* of a system $(p_0, p_1, \ldots, p_n)$ to be the history h obtained by projecting an execution of the system onto the subsequence of invoke/response messages appearing in the execution.

An object's sequential specification defines its sequential behavior, and we must now define its concurrent behavior. 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. The precise definition of linearizability is well-known [HW90], so we will not repeat it here.

Finally, an implementation is said to be *wait-free* if no front end is blocked by the failure of other front ends. Specifically, for every history h of the implementation and every nonfaulty system process p_i in h, every invocation of an operation by p_i in h has a matching response.

4 The Increment Register

In this section, we give our optimal wait-free implementation of an increment register.

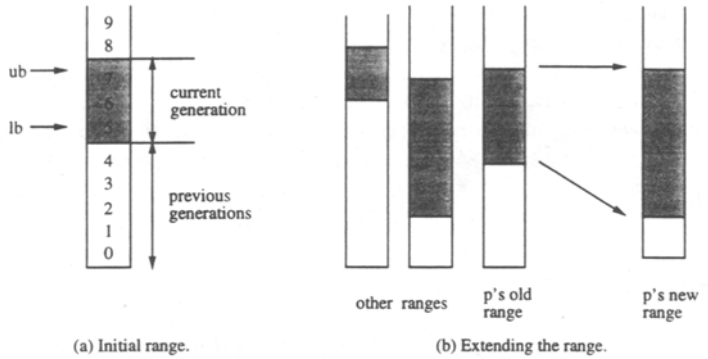


Figure 1: The range.

A process 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 processes invoking these increments as *generation k processes*. 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 $k + \ell$.

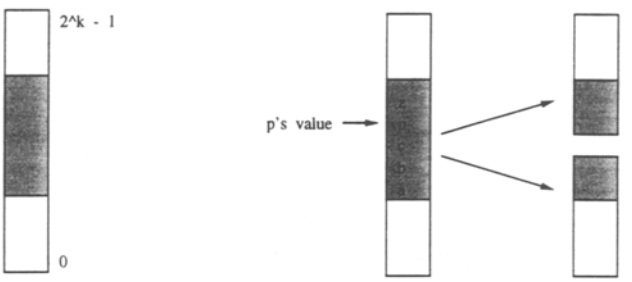
Since a process p can invoke the increment operation more than once, it identifies itself during generation k with an ordered pair $\langle p, k \rangle$ called its *increment process id*. We assume each process p maintains a set $IncSet$ of all the increment process ids that it knows about, and continues 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 processes, and merges the sets it receives from other processes 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 $\langle p, k \rangle$.

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

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 1, each process p maintains a *range* $R = [R.lb, R.ub]$ of return values. Initially, using the set $IncSet$ of increment process ids known to p, process 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, process p exchanges ranges with other processes in its generation, and extends its range by dropping its lower bound to the smallest lower bound received from any of these processes.

Intuitively, by setting its initial lower bound to lb, process p is reserving lower values $v < lb$ as return values for increments in previous generations recorded in $IncSet$. Later, if p hears that another process q in the same generation set its initial lower bound to $lb' < lb$,



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

Figure 2: The interval.

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 process sets its upper bound to ub, then all processes 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 processes partition the return values. Nonfaulty processes 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 [HT90]. 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 implement this idea, process p maintains an *interval* $I = [I.lb, I.ub]$ of return values that contains its range R (see Figure 2). The size of the interval is always a power of 2. Process p's initial interval is the smallest interval of the form $[0, 2^k - 1]$ that contains p's initial range. During an increment, process 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 2). Every round, process p exchanges intervals with other processes, and p maintains a set C of all processes sending p an interval intersecting its current interval I. The processes in C are p's *competitors* since they include the processes considering return values in p's range. To avoid returning the same value as one of its competitors, process 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 process id. Eventually, p assigns a value v to itself. Process 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, process p's interval eventually contains a single value v, at which point p chooses v

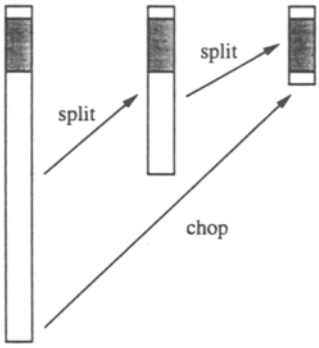


Figure 3: Chopping.

but continues exchanging its interval with other processes until all processes 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 logarithmic nature is logarithmic in the size of the initial interval, which can be as 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 3. 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, process 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 its range by lowering the lower bound of its range, 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 4, the definitions of splitting and chopping are given in Figure 5, and the definitions of some initialization steps are given in Figure 6.

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

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

  repeat
    broadcast <p,R,I,lb>
    receive   <p',R',I',lb'> from generation k processes 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 4: The increment register $\mathcal{I}$.

```
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 5: Chopping and splitting an interval.

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

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

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

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

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

Figure 6: The initialization phases.

processes, 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 process 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 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 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 in the previous section's discussion of ranges:

Lemma 1: Suppose p and q are generation i and j processes returning values v and w, respectively. If $i < j$, then $v < w$.

For the 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 process to assign values to its competitors (including itself). The second property we must prove is that $|C| \leq |E|$ for every process 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 process 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 processes 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 $|C| = 2 \not\leq 1 = |E|$.

Proving that $|C| \leq |E|$ requires reasoning about the interactions between the splits and chops performed by different processes in different phases, and we prove two claims (Claims 4 and 5 below) about these interactions. Let us fix a generation k for the rest of this paper. 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 2: If p splits from I_{i-1} to I_i, then the upper bounds of $R_i \subseteq E_i \subseteq I_i$ are equal if p splits down, and the lower bounds are equal if p splits up.

Fact 3: If p chops from I_{i-1} to I_i, then the upper bounds of $E_{i-1} = R_i \subseteq E_i \subseteq I_i \subset I_{i-1}$ 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 process p's splitting and chopping usually involves reasoning about another process q's behavior in earlier phases. The first claim below argues that whenever a process 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 4: If $I_{q,j} \supseteq I_{p,i}$ for some $j < i$ and $I_{q,j}$ appears maximal to q in phase j, then $C_{q,j} \supseteq C_{p,i}$.

Proof Sketch: If r sends p an interval intersecting $I_{p,i}$ in phase i, then r sends q an interval intersecting $I_{p,i}$ and hence intersecting $I_{q,j} \supseteq I_{p,i}$ in the earlier phase $j < i$. □

The second claim we prove concerns the fact that a process p may split into an interval I in an orderly sequence of splits while another process 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 5: 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 Sketch: If p splits down from $\hat{I}$ to I, then q chops down into I, and its new interval J has the same lower bound as I. Since p splits down, its interval I and range have the same upper bound. Since q chops down, its interval J and range have the same lower bound, so p's interval I and *extended* range will have the same lower bound at the end of that phase.

If p splits up from $\hat{I}$ to I, then q must chop up into I. Since p splits up, its interval I and range have the same lower bound. Suppose, however, that the top of its range is lower than the top of its interval. We can argue that p's initial range must extend below I (or p would have chosen I as its initial interval), so q's early ranges must extend below I, so q would never be able to *chop* up into I, a contradiction. □

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 processes 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 process 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 6: If R contains some process q's initial range $R_{q,1}$, then $|C| \leq |E|$.

Claim 7: If I contains some process q's initial interval $I_{q,2}$, then $|C| \leq |E|$.

As for the inductive step itself, if I does not contain the initial interval of any process, then all of p's competitors have chopped or split into I. The next result concerns the chopping case. It says that if I is p's interval and if any process 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 8: Suppose $\mathcal{I}_{\ell-1}$ is true. If any process has chopped into I by phase ℓ, then $|C| \leq |E|$.

Proof Sketch: If I contains the initial interval of any process, we are done by Claim 7. If p chopped into I in phase $i \leq \ell$, then $C \subseteq C_{p,i-1}$ and $E_{p,i-1} \subseteq E$, and the result follows from $\mathcal{I}_{\ell-1}$. If p split into I, then some process q chopped to $J \subseteq I$ in phase $j \leq \ell$. We can prove that $C \subseteq C_{q,j-1}$ and $E_{q,j-1} \subseteq J \subseteq I = E$, and the result follows from $\mathcal{I}_{\ell-1}$. □

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 our increment register algorithm differs from the strong renaming algorithm it is based on.

Claim 9: 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 Sketch: Let q be the greatest competitor in C, meaning q is the greatest process 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 4. Since q is the greatest process in C and since q split down from $\hat{I}$ to I, process q found that all processes in $C \subseteq C_{q,j}$ could choose distinct values from the bottom half of its extended range. Since E is at least this big, we have $|C| \leq |E|$. □

Claim 10: 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 Sketch: Let q be the least competitor in C, meaning q is the least process 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 4 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 processes p and q receive messages from exactly the same set of processes 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.

Prove that $E_{e',q,i-1}.lb = E_{e',p,i-1}.lb \geq E_{e,q,j-1}.lb$ and $C \subseteq C_{e,q,j-1} \subseteq C_{e',q,i-1}$, meaning that at the end of phase $i-1$ in e' process q's lower bound is higher and set of competitors is larger than 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, since p and q have extended ranges with the same lower bound and receive the same intervals at the end of phase $i-1$ in e', they must assign the same values to the same processes 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 processes in $C_{e',p,i-1} = C_{e',q,i-1}$, and we have already noted that they assign the same values. Since q is the smallest process in $C \subseteq C_{e',q,i-1}$ and q splits up, this means that both p and q can find values for all processes 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. □

The invariant $|C| \leq |E|$ follows by induction on ℓ, and the correctness of our implementation follows by this invariant and Lemma 1:

Theorem 11: $\mathcal{I}$ is a linearizable, wait-free implementation of an increment register.

6 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.[2] Our algorithm has the nice property that the invocation of an increment operation delays at most one generation. If the invoking process is nonfaulty, then the increment delays its own generation. If the invoking process is faulty, then it may delay

[2]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.

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 processes, namely those processes or invocations that contribute to the generation's running time. We show that the largest range chosen by any generation k process 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.

6.1 Active Sets

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

Loosely speaking, the active set for generation k consists of all processes that the "good" processes learn about for the first time in round k. Remember that all processes 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" processes for generation k are the generation k processes that survive these initialization phases and begin broadcasting intervals.

Let gen_k be the set of generation k processes. Formally, we define $good_k$ to be the set of generation k processes 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 process p, the set of processes 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 processes the good processes 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 processes that the good processes 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 processes increases with every round:

Claim 12: $known_{k-1} \subseteq known_k$ for all k.

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

Claim 13: $good_k \subseteq active_k$ for all k, and $active_j \cap active_k = \emptyset$ for all $j \neq k$.

6.2 Maximal Range

For each generation k, we can bound the size of the ranges sent by good processes 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 processes, since all other processes fail by the end of phase 1.

Consider the largest range a good process p can send. Every process p chooses upper and lower bounds u_p and l_p at the end of phase 0, but then p decreases its lower bound in every round. At any given time, a process p's lower bound is the minimum of the lower

bounds l_q chosen by some subset of the generation k processes. In the worst case, a good process 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 14: $R_{p,i} \subseteq max\_range_k$ for every good process $p \in good_k$ 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 process in generation k.

Claim 15: $|max\_range_k| \leq |active_k|$.

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

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

since $known_{k-1} \subseteq known_k$ by Claim 12. □

6.3 Running Time Analysis

For each generation k, we can bound the size of intervals sent by good processes 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 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 $|max\_range_k| \leq |active_k|$.

Claim 16: Given any sequence of intervals $I_1 \supset I_2 \supset \cdots \supset I_l$ sent in phase 2 of generation k, we have $|I_2| \leq 2|max\_range_k|$.

Proof Sketch: Since the intervals I_i in the chain are sent in phase 2, they are sent by good processes in $good_k$ (processes surviving phases 0 and 1), and their ranges R_i are contained in max_range_k by Claim 14. 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 be at least half of I_2 by the end of phase 2. This means that $|I_2| \leq 2|R_2| \leq 2|max\_range_k|$.

Suppose I_2 is in the bottom half of I_1. This means that at the end of phase 1 the upper bound $R_1.ub$ of R_1 is in the top half of I_1, and 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—or lower—so R_1 will span the top half of I_2: $|I_2| \leq 2|R_1| \leq 2|max\_range_k|$. □

Combining these results, we are done:

Theorem 17: Every generation k increment operation completes within $O(\log |active_k|)$ rounds.

Acknowledgments: We thank Maurice Herlihy for his insights and interest.

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.

[AHS91] James Aspnes, Maurice P. Herlihy, and Nir Shavit. Counting networks and multi-processor coordination. In *Proceedings of the 23th ACM Symposium on Theory of Computing*, May 1991.

[Her86] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

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

[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.

[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.

[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.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690, September 1979.

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

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

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

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