Algorithms for Unbounded Concurrency

Bring:
Extra notes for scribe.

Today's papers:
Merritt, Taubenfeld
Gafni, Merritt, Taubenfeld

Next time (actually, this will be next Thursday—we're interrupting the normal flow with a class on Byzantine quorum systems—see handout 4.
*** Afek, Greenberg, Merritt, Taubenfeld, Computing with Faulty Shared Objects
*** Jayanti, Chandra, Toueg, Fault-Tolerant Wait-Free Shared Objects

1 Some residual remarks on adaptive algorithms for renaming, and other problems

Last time, we discussed the renaming problem, from Moir-Anderson and Attiya-Fouren.
Main result of AF gives adaptive, $6k-1$-renaming with time $O(klogk)$.
One-shot. MA also define long-lived, but their algorithm isn't adaptive (time complexity depends on N, the size of the identifier space).

Afek and Merritt have improved the main result of AF to obtain $2k-1$-renaming with time $O(k^2)$ (still one-shot).
Their algorithm is based on the AF algorithms.
In particular, they use a version of the AF shrinking network algorithm in which they depend more on using the lower right part of the network.
Too complicated to present here.

Q: What about long-lived renaming?
Q: Are versions of these algorithms useful in distributed systems?

Today: I'll say a few words about a couple of other papers I read in this area, but didn't ask you to read.
Then go on to the new material on computing with infinitely many processes.

1.1 Attiya, Fouren II, adaptive long-lived $(2k-1)$-renaming

Polynomial and Adaptive Long-lived $(2k-1)$-renaming. DISC 2000

Long-lived renaming: Processes can request (getName) and return (releaseName) names repeatedly.
They obtain an adaptive $2k-1$-renaming algorithm with $O(k^4)$ step complexity.

Now k is not the total number of participants that ever appear.
Rather, they define k in terms of maximum concurrency, specifically, *point contention*:
k is an upper bound on the number of processes that are ever simultaneously outside of their "remainder regions", when they are not involved with requesting, holding, or releasing a name.

They use another building block called "sieve".
A sort of arbiter.
Can be "reserved" for a process based on suggestions by processes that invoke the sieve.
Seems complicated, not a clean abstraction.

(The renaming algorithm works by having a process choose a timestamp, which is used for priority.
Then it tries to reserve sieves for itself and processes with smaller timestamps.
It does this until it finds that some sieve has been reserved for itself.
Gets a name from that sieve.)

What's interesting: The very strong result statement.

1.2 Afek, Stupp, Touitou

Long-lived, adaptive collect with applications.

They define Collect and ActiveSet problems, and give long-lived, adaptive algorithms for them.
These appear to be useful building blocks, for transforming other long-lived algorithms into adaptive versions.

E.g., Collect:
Supports operations Write, Collect.
Abstract spec: An automaton whose state contains N single-writer, multi-reader registers, one for each process id.
To Write, a process writes its value in its own register.
To Collect, it reads the N registers one at a time, in any order, and returns all the values read (set of (process,value) pairs), for values that have ever been written.

Their implementation is adaptive, according to the point-contention definition of AF.
It uses multi-writer, multi-reader registers, each of which can hold as many as N values.
It uses an $O(N^2)$ element linear array in which processes store their values.
In order to make it adaptive, processes try to capture an element as close to the beginning of the array as possible, for writing their values.
Also, they work on "bubbling up" their values to the beginning of the array.
Many values can be put into the first entry.
Somewhat complicated.

Applications:
Many long-lived algorithms follow a discipline where they alternately write their own single-writer

register and read everyone's register.
Can transform such algorithms using a Collect service:
Do the writes to the Collect service; replace all the reads with Collects.

Sometimes this yields an adaptive version.
Examples: The standard atomic snapshot algorithm.
An ℓ-exclusion algorithm.

Warning: The adaptive version isn't adaptive according to the usual point-contention definition.
Rather, according to a weaker definition of "interval contention".
In this definition, contention is calculated as the maximum number of operations whose intervals overlap any single operation.
(diagram)

Q: Why is this?
A: During a snapshot, k processes perform updates, one at a time.
The snapshot's time will depend on this k (because it can repeatedly restart every time there's a change). But the point contention is only 2.)

Suggests that point contention isn't such a robust notion.
What is the right notion?

ActiveSet:
A more basic service, which simply detects who appears to be currently active.
It can be used to implement a variation of a Collect object:
Use ActiveSet to determine who seems to be active, and only read those processes' individual registers.

ActiveSet supports operations join, leave, and getSet.
Abstractly, the service has N single-writer multi-reader Boolean flag registers, one per process, initially all false.
join: process writes true in its flag register.
leave: writes false.
get-set: goes around reading in any order, returns the set of processes for which the flag is true.
Nice abstraction.

They produce an algorithm with a sort of "adaptivity" property, though not in terms of point contention.
Rather, the time cost depends on the size of the current set, that is, they count in k all those that have invoked a join and haven't received a leave-ack.
Not just on the number of processes currently engaged in join and leave operations.

Example using ActiveSet: Lamport's bakery algorithm, p. 9, which they make adaptive by using ActiveSet.
This (I think) uses the point contention definition, based on the interval when the process isn't in its remainder region.

Significance: Ways of measuring contention—what are good, robust measures?
Some interesting building blocks.
Adaptive algorithms for these.

1.3 Attiya, Fouren, Gafni, adaptive collect with applications

1.3.1 Overview

This time, adaptive refers to total contention (k = total number that have ever participated).

Collect problem formulated slightly differently from before: a list of properties rather than an automaton spec.

Write: Write value in your own location.
Collect: Returns a view, which behaves as if it resulted from a global read of all potential locations.

Their implementation of this object incurs latency:
O(k) for the initial write by each process
O(1) for subsequent writes
O(k) for collect.

Nice.
Weaker than Afek, Stupp, Touitou because they don't try to make it adaptive to point contention, just total contention.
Thus, they don't "bubble up" values; just leave values where they are, once written.

Show how to use adaptive Collect to implement adaptive one-shot atomic snapshot object, "immediate snapshot" object.
Use these to get a (2k-1) adaptive renaming algorithm with $O(k^3)$ latency.
Not as small complexity as Afek-Merritt, but more modular.

1.3.2 The Collect implementation

Simple. Uses complete binary tree of Lamport (or Moir-Anderson) splitters (now using "left" and "right" instead of "right" and "down").
n nodes, where n is an upper bound on the number of potential participants.

Each splitter has X and Y plus other variables:
mark, which allows the node to be marked (when it is first visited)
id, which holds the identity of a process that stops at the node
value, which holds the latest value written by that process.

The first time a process writes, it obtains a node in the tree.
It writes there and remembers the node.
On later writes, it just goes directly to its node, overwrites.

To obtain the node in doing the first write, a process traverses the tree from the root.
At each node, it enters the splitter, and marks it.
If outcome is "stop", that is its node (it's guaranteed to be the only process that ever stops there.
If outcome is "left" or "right", it goes down that branch and continues.

To do a collect, any process does a DFS of the tree, visiting the marked nodes only.
It collects all the (process, value) pairs at all the nodes, and returns that as its view.
Satisfies the requirements of the Collect object.

Time bounds:

Note: The marked nodes are solid from the top of the tree.

Claim: At most $2k - 1$ nodes are ever marked (by at most k processes that write).

Proof: Consider a finite execution in which at most k processes have begun write operations.
Let S be the set of vertices that are marked in that finite execution.
Show $|S| \leq 2k - 1$.

Order S in in-order: for every marked vertex v, vertices in left subtree are ordered before v, vertices in the right subtree are ordered after v.

Color the vertices:
Grey: If both its children are marked.
Black: If one or zero of its children are marked.

Claim: In the (in-order) ordering of S:
1. The first vertex is black.
2. The last vertex is black.
3. No two consecutive vertices are grey.

Proof:
1. If the first vertex were grey, its left child would be marked.
But then it isn't numbered first in in-order.
2. Symmetric.

3. Assume two consecutive vertices in in-order say v_i and v_{i+1}, are grey.
If they are incomparable, then their LCA is between them in in-order, so they aren't consecutive.
So one is an ancestor of the other. WLOG, say v_i is an ancestor of v_{i+1}.
Then by in-order, v_{i+1} must be in the right subtree of v_i.
But then v_{i+1}'s left child w can't be marked: if it were, it would be between v_i and v_{i+1} in in-order.
But that contradicts the assumption that v_{i+1} is grey.

Claim: With each black vertex, we can associate some process that either stops their or fails there.

Proof: The node gets marked, so someone arrives.
Not all the entrants to the node can go to one child, by the splitter definition.

Since no one goes to the other child, someone must stop here.

So, this means that there are at most k black vertices.
That implies at most $2k - 1$ total marked vertices.

This linear bound gives a linear bound on collect operations.
Also, the first write incurs a linear time bound, and the rest are constant.
(Not logarithmic—this isn't balanced.)

1.3.3 Applications

As Afek et al., they plug their adaptive collect into other algorithms and speed them up.

Ex: Atomic snapshot: Recall the def. from my course.
But here they modify it a bit, using only a single operation that both writes a value and returns a view.
The effect is as if the view were read all at once.
However, they don't require the write and the view snapshot together to be atomic—there can be interleaving between the two.

Their algorithm is a modification of an $O(nlogn)$ snapshot algorithm of Attiya and Rachman, modified to make it adaptive by: using a tree of successive-increasing-size binary trees, and using adaptive collect.
(This trick is similar to one we saw last time, using a sequence of range-independent renaming algorithms to get an adaptive renaming algorithm.)
Get latency $O(klogk)$.
LTTR.

Ex: Immediate snapshots. Like snapshots, with upscan, except that the entire upscan operation looks atomic. (I think—their definition isn't expressed this way, but in terms of axioms.)
They give a somewhat-complicated algorithm, with latency $O(k^2)$.

Ex: One-shot, $2k - 1$-renaming, with $O(k^3)$ latency.
Not quite as good as Afek-Merritt, but done in a stepwise, modular fashion.

1.4 Discussion: Renaming, Collect, Lattice Agreement, in distributed systems

Useful? How are the problems different? How to implement?

2 Merritt, Taubenfeld

Computing with Infinitely Many Processes.
Longer version, written after the conference version, DISC 2000.
Looks like it's submitted for journal publication, or about to be.

Warning: This is less polished than the other papers we've seen so far—those are final versions of

journal papers.
Note the date. That's because I found some errors reading it, and they did some quick fixing-up.

2.1 Overview

Paper considers leader election, mutual exclusion, and consensus.

They consider both read/write registers and stronger primitives, like test-and-set, semaphores, etc.
I'll focus on read/write.

Allow infinitely many potential participants.

So far, nearly all the papers we have covered have assumed a finite bound of some n.
Exception: Friedberg-Peterson m.e. algorithm with two weak semaphores.
They assume no bound on the number of participants.
Still get no-lockout, even if more and more processes keep arriving while a process i is active.

Here, individual results assume partial info about the number of participants, e.g., at least c will participate.

Lots of results (see overwhelming-looking but helpful charts on p. 4 and 5).
Hard to get a handle on what's important.
I'll cover:
A lower bound for leader election, Theorem 2.1.
Theorem 2.4 (election with 2 registers)
Theorem 3.2 (adaptive starvation-free mutual exclusion.)
And Theorem 5.2, which relates back to Friedberg-Peterson.

All their results assume no failures.

Something funny:
Not only don't they assume an a priori bound n on the number of processes, they also say they allow infinitely many processes to be active at the same time.
This is hard to understand.
I understand infinitely many potential participants, no bound on how many may participate in the course of one execution, infinitely many allowed over the entire (infinite) execution.
But don't like to think about infinitely many processes taking steps together—an infinite amount of activity happening at once!

MT's real aim is to design algorithms that scale well.
I don't think allowing infinite concurrency at one point in time is crucial.
It certainly doesn't arise in any of the main results I've selected.
So I'll ignore this.

Concurrency level:
They consider upper bounds on the point contention (number of processes that may be active simultaneously):

- finite: There is a finite upper bound c on the number of processes that are simultaneously active.
  This bound holds for all runs.
- bounded: In each run, there is a finite upper bound on the number of processes that are simultaneously active. The bound may be different in different runs.
- unbounded: In each run, at each point, only finitely many processes are active, but there is no bound.
- infinite: (The one I don't like) Infinitely many may be simultaneously active. They talk about "group steps" involving infinite amount of activity.

Participation level:
They also consider lower bounds on how many processes *must* participate:
ℓ-participation: Restrict attention to (fair) executions in which at least ℓ distinct processes participate at some point. (So, a process can wait for that many.)

Actually, they define upper bounds on participation also, but seem not to use them in the paper. So I'll ignore.

Properties considered:

- Adaptivity: As we have been using the term. Time complexity of operations are a function of the actual concurrency. Point contention.
- Symmetric: Processes have uids, but are otherwise indistinguishable. Uids are from a set with no operations except comparison for equality only. No ordering. Cannot use uids to index registers.

Key results (Read/write registers):

1. Leader election can't be done with finite shared space. Theorem 2.1.
2. Leader election and consensus can be solved with two unbounded-size registers, assuming the number of participants is at least c and concurrency (number of concurrent participants) is at most c.
   Their algorithm is adaptive and symmetric. Theorem 2.4.
3. Mutual exclusion needs an infinite number of registers (Burns, Lynch). They give an adaptive, symmetric algorithm for mutual exclusion, using an infinite number of infinite-size registers, with no required participation, and unbounded concurrency. Theorem 3.2.

2.2 Impossibility result for leader election with finite space

Can make rather strong assumptions.
Assume a fixed number $c > 1$ of participants.
We don't assume that the process identifers are all known ahead of time—this proof is based on an infinite universe of potential participants.
We just know the exact *number* that will participate.

In their version of the leader election problem, they just require that exactly one participant be

elected as leader.
The others don't need to know who he is.

Proof: By contradiction. Assume finitely many registers of finite size. We'll construct a bad execution.

WLOG assume that each process that participates continues taking steps forever (can add dummy steps).

For each process p in the (potential) universe, we may construct an execution $solo(p)$, in which p runs alone forever.
He does not need to get elected, because that is only required if c processes participate. However, the run does something.
This execution may be infinite.

Now, for each p, in $solo(p)$, some state of the shared memory must repeat infinitely many times. Choose such a state, call it $repeat(p)$.

Also, for each p, consider the sequence of states of the shared memory that appear up to the *first* occurrence of $repeat(p)$, say $x_0, x_1, \dots x_k$.
Let $sign(p)$ be a reduced version of this that is obtained by cutting out loops in a systematic way: Start with the first state x_j that reappears and look for the latest reappearance. Delete everything after x_j up to and including that latest reappearance.
Repeat this starting after the deleted sequence.

The result, $sign(p)$, has no repeated states.
Thus, its length is bounded by the number of possible states of the shared memory.
Moreover, by the way it was constructed:
1. The first state is the start state of memory.
2. The last state is $repeat(p)$.
3. Each state differs from the next in the value of exactly one shared variable.

Since there are only finitely many such sequences and finitely many memory states, there are infinitely many processes that have the same $repeat(p)$, call it s, and same $sign(p)$, call it $\gamma = y_0, y_1, \dots, \dots$.

Now, they show that they can take any group of c of these processes and run them in a compatible way, interleaving them so that they all think that they are running alone.
But since there are really c of them, the requirements of the problem say that exactly one of them must elect itself the leader.
This sounds dangerous, since it doesn't see any evidence at all of the other processes...

To get an actual contradiction (the authors don't do this work):
Let $p1, p2, \dots, pc$ be c processes, and say WLOG say $p1$ elects itself the leader.
No others do.
Then run $p1, \dots, p(c-1), p(c+1)$.
Again, $p1$ elects itself the leader (looks the same), and no others do.
Now consider $p2, \dots, pc, p(c+1)$.

None of these elect selves (looks same).

Now, how to construct the compatible runs?
The main idea is to maneuver the shared memory to state s and simultaneously get every process to the point where state s first occurs in its solo run.
Once this is done, we can let the processes take turns running, each starting from s and ending in s.
Then all can run forever, thinking they are alone.

The trickiest part is the initial portion—getting the processes to state s.
For this, we use the fact that they all have $sign(p) = \gamma = y_0, y_1, \ldots$.
We proceed step-by-step through γ, keeping the processes synchronized.
Start with everyone in their start states, the memory in $y_0 = x_0$.

Now start with p_1 and let it run just up to the point in its own execution where it last leaves the memory in state y_0. At that point, it is about to do a write that would send the memory to y_1.
This may involve doing a lot of reads and writes, but the final memory state gets back to y_0 again.
(Any intervening loop was cut out of $sign(p_1)$.)
And p_1 has done any reads that precede the actual write step that would produce y_1.
Thus, p_1 is poised, ready to write the variable that changes in going from y_0 to y_1.

Now run p_2 similarly, to get it to the point in its own execution where it last leaves the memory in state y_0.
Then it is also poised, about to write the same variable, the one that changes in going from y_0 to y_1.
Note that it is about to write the same value as p_1, because the state y_1 determines this new value.
Repeat for all the other processes.

So, now they are all poised to write the same thing into the same variable, to convert y_0 to y_1.
Now we'll move them along to where they are about to do the write to produce y_2.
First run p_1: it has to start with its (blind) write, which changes the memory state to y_1.
Then run it from y_1, possibly back to y_1 again, and then any reads, and stop it just before it makes the transition from y_1 to y_2.

Now run p_2.
Note that it starts in y_1, whereas it expects to start in y_0.
But (this is the key) y_0 and y_1 differ only in the value of one variable, and that's what p_2 is about to write anyway (with the right value).
So, once p_2 does that first write, p_2 can't tell that it didn't start from y_0.
Let p_2 reach the point where it is about to convert the memory from y_1 to y_2.
Repeat for the other processes.

Repeat for all steps.
Eventually gets the memory to state s, and everyone to where they would be when they first get the memory to state s in their solo executions.

2.3 Algorithms for leader election and consensus with at least c participants and at most c concurrency

Now assume that we have at least c participation and at most c concurrency.
This isn't the same as saying exactly c participants.

As before, we don't know ahead of time who the participants are.
First, Theorem 2.2 describes a nonadaptive solution to leader election with an infinite number of read/write Boolean registers.

It's also not symmetric. In fact, they assume that process indices are natural numbers starting from 1, and they use the indices for indexing registers.

Proof: Use Boolean registers $b[i]$, where i is a natural number ≥ 0. All initially 0.
Process i: First reads $b[0]$.
If 1, knows leader already elected and terminates.
Otherwise, process i sets $b[i] := 1$.
Then scans the array starting from $b[1]$ until it notices that c bits have been set (may have to scan repeatedly, dovetail).
Once it sees c bits have been set, it sets $b[0] := 1$.
If it sees that its own id is the smallest among those c, elects self leader.

Note that the c bound on concurrency implies that the $c + 1$st process doesn't arrive until one of the first c participants leaves.
But no one leaves until $b[0]$ has been set to 1.
That means latecomers won't ever set their bits, and so won't confuse anyone else in determining if they are the leader.

This result relies heavily on participation from at least c processes, though it does not depend on knowing a priori which these are.

Not adaptive: The time complexity isn't just a function of the number of total participants, or the number of concurrent participants.
Rather, it depends (at least) on the maximum identifier of one of the first c participants.
Maybe worse.

Now jump to Theorem 2.4.
This gives an election algorithm for at least c participants, at most c concurrency, which is both adaptive and symmetric.
Furthermore, it uses only two read/write shared variables.
Both of unbounded size.

Much more interesting.

Adaptive here means that the time bound depends only on the number of actual participants.
Start counting time after all the required c participants have begun.

Registers:
Union, holds a set of at most c process identifiers.
(Leader,Marked), holds a process id and a flag.

The first c participating processes use Union to collect all of the first c participants' identifiers. Each such process i executes a loop where it reads Union and compares it to the set of ids it already knows locally (in $local-union-2$, which is initialized to $\{i\}$).
On each comparison, it updates both Union and lu2 to be the union of the information read and the local info remembers.
(It doesn't write Uniont if it has nothing new to write.)
This phase ends when a process sees c different ids in Union.

The second variable holds an identifier of a proposed leader, and a Boolean mark.
When the mark is set to 1, it means that the leader has been chosen (and it is the process whose id is in the Leader field).
The leader chosen is simply the last participant to write the Leader field.

By the concurrency bound, the $c+1$st participant doesn't enter until after one of the first c leaves.
By then, Marked is set to 1.
So no "latecomer" enters the main loop, modifying Union, etc.
They just read the final result.
(This mechanism is similar to the use of $b[0]$ in the simpler nonadaptive algorithm above.)

Go over code, p. 9

Start by checking Marked. If 1, skip the loop and return.
If 0, engage in the main loop.
Repeatedly read Union into $local-union-1$, check to see if it contains at least c elements.
If so, done.
If not, set both Union and $local-union-2$ to the union of your info and what you've read.
Repeat.

If at any time, you see that Marked is already set, you stop doing the loop.

Correctness:

No more than one leader elected:
This is written to actually return the leader.
(Technically, we only need to tell the leader itself.)
It suffices to show that no two processes return different leaders.

Claim that only the first c participants can write into Leader.
That's because the next participant doesn't arrive until after Marked has been set to 1 (concurrency limitation).
And then any latecomers skip the assignment to Leader.

Claim that the last of the c participants to write to Leader is the only one that can be returned as

a leader:
By the time the first process gets past the loop, all c early processes have written to Union, which means they have already set Leader.
After that, Leader no longer changes.
So any process that returns gets its return value by reading the Leader field after this point, so gets the same value.

Termination:
The argument as written isn't so clear.
Here's another attempt.

Basically, suppose someone doesn't terminate.
If Marked is ever set to 1 he would terminate, so this never happens—Marked stays 0 forever.
Eventually all the c processes arrive.
If any of them got past the loop, then eventually the Marked bit would be set, so it must be that all c continue looping forever.
We have to argue that eventually they all learn each other's information, by their discipline of reading the Union variable and writing back what they know.

Suppose not. Then eventually all the lu2 variables stop increasing (they can only increase), in a situation where not everyone knows about everyone.
After this point, no one ever learns anything new by reading the Union variable (or the lu2 variable would increase).
So from that point on, the only thing anyone writes is the value of its lu2 variable.

Define a process to be "minimal" if its lu2 set is not a proper superset of anyone else's lu2 set.

Claim that no minimal process can write more than once after the "stabilization point".
Suppose minimal process i does write twice.
That means that in between i's first and second write, process i has to read Union and observe a set that is missing some process index that is in i's lu2 set.
We know that this set can't contain anything not in i's lu2 set.
So, the set observed has to be a proper subset of i's lu2 set.
But after i's first write, such a proper subset could not have been written by anyone, because i is minimal.

So, go far enough out in the execution to where all the minimal processes have finished all their writing.

After this point, any minimal process i continues to read, infinitely often.
Every time it reads, it has to see its own lu2 set in Union (can't see any elements not in its lu2 or it would learn something new, and can't see anything missing, or it would write).
Since the minimal processes have stopped writing, but keep seeing their own lu2 sets, it means that one of these sets must persist in the Union variable forever.

But that set is missing some index.
The process with that index eventually writes (because it has something that isn't in Union).
Contradicts the persistence of this smaller set.

Latency bound (adaptivity):
We need to argue not just termination, but termination within time that's a function of the number of participants.
This should work, should depend on c. LTTR.
(The time must be measured from when the last of the c participants arrives.)

They claim that the algorithm leads to a consensus algorithm, with the same kinds of properties.
Just elect a leader and choose its value.
Of course, this is non-fault-tolerant, and consensus without failures is pretty easy in general.