Adaptive Algorithms for Renaming and Other Problems

Bring:
Extra lecture 2 notes for scribe.
A few copies of today's papers.

Today's papers:
Moir, Anderson
Attiya, Fouren
and a bit on the other renaming papers.

Next time:
*** Merritt, Taubenfeld. Computing with infinitely many processes. Sections 1-3, can skip Theorem 2.5 and stop after 3.2.3.
*** Gafni, Merritt, Taubenfeld. Can read the whole paper (it's only 9 pages, extended abstract), but I'll focus on Theorem 3.1, and not say much past Section 3.1. The rest seems worthwhile too, but I'm trying to limit time...

And then (the schedule is slipping a bit already):
*** The two papers on computing with faulty objects.

Get scribe.
Collect topics students want to cover.

1 Moir, Anderson

1.1 Overview

This paper and the others we'll discuss today involve the problem of "process renaming".

Assume we have a universe of processes, each of which has a UID from a large namespace. Say, with N possible identifiers.

A subset of the processes may want to engage in an algorithm whose costs depend on the size of the name space.
Then it would be useful to give the processes new, "smaller" names, that is, names chosen from a smaller namespace, say, one with M possible identifiers.

Like hashing.
But, we want to make sure that we have an absolute guarantee of uniqueness of names.

A fairly large literature has been devoted to solutions to this problem, both in distributed and shared memory settings.

Moir and Anderson's contributions:

They work in the shared memory setting.
Consider both reads/writes and more powerful primitives.
We'll just cover the reads/write algorithms.

Three algorithms:

1. An algorithm that allows k processes to choose names from a namespace of size $\Theta(k2)$.
   In this paper, it is assumed that k is known, though in subsequent papers, the number k is allowed to be different in different executions, and unknown.
   The algorithm is wait-free.
2. An observation that this algorithm can be combined with other work (by Borowsky and Gafni) to get down to a namespace of size $2k - 1$.
   This is known to be optimal, for wait-free algorithms.
   Details don't appear here, though.
3. They define a new extension of the problem to a more dynamic setting: names can be acquired and released over time.
   This is called "long-lived" renaming.
   It has been studied a lot since this paper.

   They give an algorithm for this case also.
   The reduced namespace now has size $\Theta(k2)$.
   However, the time complexity is large, $\Theta(Nk)$.
   (Later work has improved this, yielding dependency on k only.)

The long-lived renaming problem is a kind of resource allocation problem.
Useful for giving out "tickets" to engage in an algorithm that allows only a bounded number (M) of concurrent participants, but where the participants might change over time.
E.g., M-process mutual exclusion algorithm.
Different processes might request the resource at different times.

1.1.1 The main algorithm, reducing the namespace from N to $O(k2)$

For one-time renaming.

Namespace has $\Theta(k2)$ elements.
(Recall k is assumed to be fixed, known.)

The algorithm is constructed using a basic "building block" called a "splitter".
Depicted on p. 5, Figure 2.
When at most n processes invoke this bblock, they get divided into three groups.
(This holds for any n—n isn't built in.)
At most 1 will stop.

At most $n-1$ will "go right".
At most $n-1$ will "go down".

It follows that, if only one process accesses the bblock, then it stops.
($n = 1$, so 0 go right or down.)

The code in Figure 2 is an example of how the splitter can be implemented.
Use two variables X and Y.
Set X to your own index, test Y.
If $Y = true$ then go right.
Else set $Y := true$ and test X.
If X is still your index then stop, else go down.

This should look familiar.
It's the same structure used in Lamport's algorithm.
Moir and Anderson have abstracted this out.
Lamport also splits processes into three groups—those that win right away, those that win after a delay, and those that restart.
MA generalize and abstract from this.

Why does this satisfy the required conditions?
At most $n-1$ go right:
Because for anyone to go right, someone must have previously set Y to true.
But once someone sets Y to true, they don't go right.

At most $n-1$ go down:
The last process p to set $X := p$ doesn't go down because it can't see $X \neq p$.

At most 1 stops:
That's the mutual exclusion argument, which is a little more involved. It's the main content of Lamport's argument.

If only one process appears, then clearly he goes through the fast path and stops.

Viewed this way, a splitter divides processes into smaller groups.
Also, at most one process stops at a given splitter.
If the splitter has an associated name, that name can be assigned to a process that stops there.

How to arrange splitters to solve renaming?

Use $k(k-1)/2$ copies of the splitter arranged in a triangle, see Figure 3, p. 6.

Associate names with the splitters in the triangle as shown (there are other ways, e.g., Attiya-Fouren use one).

A process wanting a name starts at the 0 splitter in the upper left hand corner and traverses the splitters until it stops or runs off the edge.

Then it takes the name where it stops or runs off.

It traverses the splitters by invoking each in the usual way, and accepting the response received.
If it's stop, then stop.
If right, go to the next splitter in the row.
If down, go to the next splitter below in the column.

Look at code, p. 7, figure 4.

Now we have X and Y for each splitter.
The rest of the code is obvious (but the comments have reversed down and right).

The proof is done quite carefully, using invariants.
In fact, the invariants seem to add good insight.
(We've come a long way in proof standards since 1987.)

E.g., consider I7.
It says that if one of the Y variables, say $Y[r, c]$, has been set to true, then it must be that some process is either still there having set it, or is in the same column at a row below.
(The particular process that set $Y[r, c]$ to true might move below and then right, but that would be because someone else set another Y variable in the same column c to true.
By induction on the number of steps, that or another process is below in column c.)

This helps to argue that not everyone goes right.

Also consider I8.
It is used to say that not everyone goes down.
It says that if one of the X variables has been set by a process q but is no longer equal to q, then there is some other process p that is in the same row, either at the same splitter and having already reset the X variable, or to the right in the same row.

Informally: Suppose p is the last process to set X.
Then p is either still at the node, or else has seen $Y = true$ and gone right.
It can't go down because that would mean seeing $X \neq p$.

These two invariants tell us that not everyone who reaches a node goes right, and not everyone goes down.
These two facts can be used to bound the number of processes that reach any particular splitter (I9).
At most k reach the upper left splitter (0th diagonal).
On the first diagonal, each gets at most $k - 1$.
The second, at most $k - 2$, etc.
...
The $k - 1$st, at most 1.
That's what gives uniquesness of names.

How do we get the limitation?

I think the best way to see it might be to use a stronger invariant, limiting not only how many can arrive at each individual splitter, but at each "interval" of a diagonal of splitters.

Each on ℓth diagonal has at most $k - \ell$.
Each interval of size d on the ℓth diagonal has at most $k - \ell + d - 1$.

Example, p. 6, figure 3.
Here, $k = 5$.
Max number at upper left is 5, then next diag is 4, etc.
Consider, e.g., the node numbered 6. It's supposed to have at most 3, according to the count.
See this because the two nodes 1 and 5 together can have at most 5, and each separately can have at most 4.
Case 1: If neither 1 nor 5 has 0, then each "leaks" at least one, so the total of 5 reduces to 3 when they send to node 6.
Case 2: If node 1 has 0, then node 5 has at most 4, is the only source for node 6, and leaks at least one. Again, at most 3.
Case 3: Node 5 has 0,...similar.

This can be extended to a general argument, for any interval of any diagonal.
Work by induction on the number of the diagonal.
Consider an interval of length d of diagonal ℓ.
Look at the possible sources, in the previous diagonal $\ell - 1$.
That's an interval of length at most $d + 1$.
So the number of processes that can reach there, by assumption, is at most $k - (\ell - 1) + (d + 1) - 1$.
That's 2 more than our goal.
But at each end, either leak at least one or don't have any at all.
If it doesn't have any at all, the interval of possible sources is actually smaller by 1, which reduces the number of processes that can be there by one.

In their algorithm, they don't actually include the final diagonal.
So the number of splitters is slightly smaller than the number of names.
Still $\Theta(k2)$.

Running time: Each process takes at most $O(k)$ steps from when it starts until it gets a name. So, $O(k)$ time.

Note: This wasn't written to sound "contention-sensitive". But it should be apparent from the description that it is.
Namely, suppose that the grid is infinite in the right and down directions.
Suppose that k isn't a fixed upper bound, but just the number that happen to arrive. (Some other number, n, may be a fixed upper bound, whereas N is the total number of identifiers.)
Then the same algorithm will work, allocating names from the upper left triangle to processes that happen to arrive.
$O(k2)$ names are used if k processes happen to participate.
Time still $O(k)$.

1.1.2 A corollary, reducing the namespace from N to $2k-1$

The renaming problem has the interesting property that it can be applied repeatedly, reducing a large namespace to a small one in several stages.
Using different algorithms for different stages can sometimes produce an overall improvement.

Here, they refer to an algorithm I didn't go back and read: a renaming algorithm by Borowsky and Gafni, from a PODC 93 paper.
It allows k processes to reduce the size of the namespace from N to $2k-1$ (the theoretical minimum).
The time complexity, however, is $\Theta(Nk2)$; thus, it depends on the total size of the namespace, which isn't so good.
(I don't know the algorithm, LTTR.)

MA simply observe that if the namespace size N in BG is a function of k anyhow, then this brings down the time complexity.
So, first use the MA algorithm as above, where k processes get down to $k(k+1)/2$ names in $O(k)$ time.
Next, the processes use BG's algorithm, getting from $O(k2)$ names to $2k-1$ names in $O(k4)$ time.
The net result goes from N to $2k-1$ names in $O(k4)$ time.

Q: Does this combination also yield contention-sensitivity?
That is, suppose we have an overall bound n on number of participants but in a particular execution we may have fewer.
Then the MA algorithm works in time $O(k)$ as noted above.
However, BG is a problem.
It has the upper bound n built in, for example, it involves taking snapshots by reading from all n processes' shared variables, one at a time.

Later papers, by Attiya, Fouren, and others do obtain $2k-1$ with real contention-sensitivity. (Time depending on k, not n.)

1.1.3 Long-lived renaming algorithm

Long-lived renaming is formulated like mutual exclusion, in terms of "regions": remainder, getname (trying), working, and putname (exit).
They assume a fixed k. Now it's used as a bound on how many processes can be active (anywhere except in remainder region) at once.
Uniqueness condition says that no two processes are granted the same name at the same time.

The algorithm isn't completely satisfactory because it has time complexity $O(Nk)$.

Algorithm uses the same kind of grid, but modifies the splitters to allow the Y variables to be reset when a process finishes with that splitter.
Now, this introduces some race conditions—between a process that sets Y and one that resets it

back to *false*.
To avoid this, MA are forced to replace Y with a collection of single-writer variables, one per potential participant (that's N of them).
Thus, instead of setting the common Y, everyone sets its own $Y[p]$.
And unsets it when done with this splitter.
The cost of this is that, instead of reading one Y variable, everyone has to read N of them to make sure no one else is conflicting.

(see code, p. 14, Figure 5)

Note lines 3, 4, 6, and 8, where the new Y bits are used.

The proof seems to follow that of the non-long-lived algorithm pretty closely.
The key invariant is now I25, on p. 14.
An analog to I9.
It essentially says that, in any reachable state, the total number of processes that "have entered a particular splitter and haven't yet left the system" is bounded by the same number as in the previous proof.

All this yields $k(k+1)/2$ namespace, $O(Nk)$ time.

What kind of contention-sensitivity bound does this exhibit?
Not too good, because of the need to read N variables at each splitter.

1.1.4 Other results

These use stronger primitives.
Not so widely studied.
They are derived mostly from multiprocessor memories.
Can they be useful in distributed systems? Mark M. thinks so.

1.2 Significance

Contention-sensitive algorithm for renaming, with $O(k2)$ names and $O(k)$ time.

A way of combining this with a prior algorithm to get optimal namespace, but time complexity isn't contention-sensitive.

Defined the long-lived renaming problem, but the algorithm given doesn't have good contention-sensitivity.

Renaming problem seems of some use in distributed systems.

1.3 Work this suggests

Design algorithms for which the size of the resulting namespace and the time complexity depend only on the number of processes that actually show up (or, that show up at the same time).

Try to attain optimal size of namespace, while time depends on who shows up.

Consider what variants of the renaming problem make sense in distributed settings, and for what applications.
See which algorithms from the shared-memory setting may carry over.

2 Attiya, Fouren

2.1 Overview

A widely-cited paper in the area of contention-sensitive algorithms.
They coin the term "adaptive" to capture contention-sensitivity, and this term has stuck.
They do a good job of setting out basic definitions for the concepts.

Consider renaming and another problem, "lattice agreement", which is a kind of membership problem.
Using read/write shared vars.
Everything is "one-shot", not "long-lived" as at the end of MA.

New results:

1. An algorithm to achieve $2k - 1$ renaming (k is the number of actual participants), with time complexity $O(N)$ (N is the size of the total identifier space, as before).
   This beats Borowsky-Gafni's bound.
2. An algorithm to achieve $2k-1$ renaming with time complexity $O(nlogn)$, where n is an upper bound on the number of participants.
3. An algorithm to achieve $6k - 1$-renaming with time complexity $O(klogk)$.
   This beats MA because it gives linear namespace and poly time complexity, both in terms of the number of actual participants.
4. An algorithm for "lattice agreement" with time complexity $O(klogk)$.

In addition, they show various ways that their algorithms can be used as building blocks for other algorithms.
A good start at a modular theory for this area.

2.2 Definitions

Three parameters: N, the total number of identifiers; n, an upper bound on the number of possible participants; and k, which is used to represent a bound on the number of participants in a given execution.

Adaptive: Step complexity (a time measure) is bounded by a function of k.

Range-independent: Step complexity bounded by a function of n.

2.3 Moir-Anderson revisited

They reformulate this to allow an unbounded grid.
They enumerate the splitters across diagonals rather than row by row.

2.4 $2k-1$-renaming in $O(N)$ steps

Network of "reflectors".
See diagram top of p. 8.

A reflector has two entrances, four exits.
If you enter on in0, you exit either on up0 or down0.
Similarly for in1.

Network connected as shown (go over).
Has one column per element of N (the huge namespace).
A process with name c starts at the top of column c and heads downward, as long as it doesn't encounter evidence of any other process.
When it does, it starts moving right, at first heading directly right, in the same row.
As it continues to move right, it changes row by moving one row up or down at each step.
The final name is the name of the row on which it exits the rightmost column.

Key property: (Proved by induction on the column number):
No two processes exit column c headed for the same row.

When a process with a number smaller than c enters column c from the left, then:
If it doesn't see that the process with name c has already passed it going downward, then it heads diagonally one step downward (takes exit down0).
If it does see that the process has passed, then it heads upward (takes exit up0).

E.g., notice that if process c heads right in a particular row, to column $c+1$, then anyone hitting column c in that row will head to the row directly above or directly below—not the same row as process c.

See p. 10, figure 4(c):
Basically, this figure says it all.
Processes that arrive from the left and see that process c (q, here), has passed all head up.
Those that see that process c hasn't passed all head down.
q itself heads straight across.

It doesn't seem completely obvious that this nice picture of the behavior can be created independently of timing.
It helps to realize that q heads downward until it hits evidence that some process has come by (and moved downward).
At that point, q moves directly right.
Anyone who hits column c above this point, at any time, must arrive after q passed on the way down. So they will all go up.

Anyone who hits column c below this point, at any time, will not see q. So they will all go down.

The names assigned are the row numbers.
The unique rows lemma means that all the processes get unique names.

How large is the namespace?
If k processes enter, they will all proceed downward and occupy the bottom part of the triangle.

But one process can "waste" two rows:
E.g.: When q going down in col. c hits row r and finds that p has already been there, then q goes right, hitting col $c+1$ in row r.
p has already gone diag down to hit col $c+1$ in row $r-1$. And no one will ever hit col $c+1$ in row $r+1$.
(No one else enters col c at row r and anyone entering above goes upward upon hitting col c.)

Exercise: Prove that only the bottom $2k-1$ rows can end up being occupied, at the right col.
(An induction on number of cols, I think.)

How is the reflector implemented?
Easy: See p. 9, algorithm 2 code.
Uses two Boolean registers R_0 and R_1 to represent the two entrances.
Someone who enters on an entrance sets the register for that entrance, then tests the other register.
Note the race condition:
We can have one arriving and seeing the other false, and going down, then the second one goes up.
As we've been discussing.
But we can also have both arriving at the same time and both going up.
Doesn't upset anything we've said.

So, Lemma 3.3, p. 8, is the lemma about unique rows.
Inductive proof.
Figure 4, p. 10, describes the inductive step.

Theorem 3.5, p. 11:
Solves $2k-1$-renaming, with step complexity $O(N)$.

The $O(N)$ describes the time to traverse down and then right.

2.5 $2k-1$-renaming in $O(nlogn)$ operations.

Recall: n is an upper bound on the number of possible participants.

Actually, I'm not so fond of this result.
Instead of presenting it, I'll describe a simpler one that is almost as good—yields $O(n2)$.

Combine Moir and Anderson's $k(k+1)/2$-renaming algorithm with thte non-adaptive $2k-1$-renaming algorithm just presented.

Namely, first reduce the names into a range of $n(n+1)/2$ using Moir-Anderson.
(Where n is the static upper bound.)

Then use the resulting names to enter a shrinking network as described above, with (statically defined) $n(n+1)/2$ columns.
This reduces the names into a range of $2k-1$, with running time $O(n2)$.

The whole thing yields a $2k-1$-renaming algorithm with $O(n2)$ running time.

Reducing the running time to $O(nlogn)$ involves a somewhat-complicated binary tree of shrinking networks.
LTTR.

2.6 $6k-1$-renaming in $O(klogk)$ steps

This uses the previous result.
Since I gave a weaker version for that, I'll give a weaker version here also, which will yield $6k-1$-renaming in $O(k2)$ steps.
The important thing is that it's completely adaptive.
Linear size namespace, within factor of 3 of optimal, and takes time depending on number of actual participants.

Uses lattice agreement.
What's lattice agreement?
A service that can be invoked by some set of processes (with names from N).
Get a response that is a set V of processes that have performed invocations (a "view").

Two simple requirements:

1. Any process is included in the view it receives (*self-inclusion*).
2. The set of all the views returned is totally ordered by containment.

They give an algorithm based on one of Inoue et al., 1994 WDAG.
Wait-free, step complexity $O(klogk)$. Adaptive, therefore.

Come back to this in a minute.
For now, assume this and use it for renaming.

The algorithm uses one adaptive, wait-free lattice agreement service, LA with step complexity $O(klogk)$, and a sequence $RIR_1, RIR_2, RIR_4, RIR_8, ...RIR_{\lceil logn \rceil}$ of wait-free range-independent $2k-1$-renaming services with step complexity $O(n2)$, as described above.

A process wanting a new name first invokes LA to get an approximation to the set of other processes who are around.
The *number* of such other processes leads it to choose one of the RIR services.
Does this by rounding the number up to a power of 2, say 2^j.
Then it invokes RIR_{2^j}, which returns a name.

Let this be a "temporary name".

Convert it to a final name by using disjoint namespaces for the various RIR services.
Service RIR_{2^j} uses the names in $(2^{j+1}, 2^{j+2}]$, e.g., RIR_4 uses names $9, 10, 11, ...16$.

LA guarantees that the number of processes that engage in RIR_{2^j} is at most 2^j.
So for this service, $n = 2^j$.
The output namespace needs at least $2(2^j) - 1$ elements, and the range given above, $(2^{j+1}, 2^{j+2}]$, has enough.

Show this yields $6k - 1$-renaming:
I'll first argue $8k$ because it's simple; the rest involves a little special case analysis.

If k processes enter, the largest set returned by LA has size k.
Taking $j = \lceil logk \rceil$, we see that RIR_{2^j} is the largest RIR that is ever run.
The largest name that is ever used is 2^{j+2}.
Since $j = \lceil logk \rceil$, this implies that the largest permanent name is at most $2^{\lceil logk \rceil + 2}$
Which is at most $8k$.

The $6k - 1$ comes about by noticing that the number of names used in the last, largest range is at most $2k - 1$, rather than the whole range, which has size $2^{\lceil logk \rceil} + 1$ elements.
So we really need only $2k + 1 + 2^{\lceil logk \rceil + 1}$ names.
That's at most $2k - 1 + 4k = 6k - 1$.

Time complexity:
$O(klogk)$ for the adaptive LA.
Then $O(n2)$ for the RIR. Here, $n = 2^j$ where $j = \lceil logk \rceil$.
So this yields $O(2^{2j}) = O(2^{2\lceil logk \rceil})$.
This is $O(k2)$.

Summarizing:
The key idea here is to convert a range-independent algorithm to an adaptive one by using estimates of the number of participants.
The estimates come from a lattice agreement "subservice".
They allow partitioning of processes according to sizes of estimated membership views.
Each partition component has a fixed size, which allows the members to participate in a range-independent algorithm.
For some problems, like renaming, the results can be combined.

Q: What happens if we don't do the powers-of-two trick, but just use the size of the set that comes from LA as the static n for a RIR?

A: Too many names, when we make the ranges disjoint. Becomes quadratic again.

We could also partition by powers of 3, or 4,... instead of 2, and do somewhat better.
In the limit, this gives a $4k - 1$ bound.

2.7 Lattice agreement in $O(klogk)$ steps

Now, how to implement lattice agreement adaptively?

This is based on an unbalanced binary tree, depicted on p. 16.
At each node of the tree, a "union" component appears.

What union does:
Invoked by a set of processes, some on left-child port and some on right-child port.
Invocation includes a view (set of processes) parameter.
Assumes that all the invocation views on each child separately satisfy self-inclusion and total ordering by containment.

Response to each process is a view whose members are all included in some input views.
Moreover, the output to each process includes its input view.
And all the output views are totally ordered by containment.

Union can be implemented by a wait-free adaptive algorithm, with time complexity $O(k)$.

I don't have time to go over the union procedure.
It's in appendix A.
It assumes a fixed n.

Basic idea: An array $Views_0$ of views is kept for side 0 and another for side 1.
A process that has an input view with ℓ elements writes that view into entry ℓ of the appropriate array.
(The nesting assumption implies that the entry is unique.)
Then the process reads both arrays according to various tricks, until it finds a "stable" max size view from both.
(Does repeated reads, alternating, until it sees that both max views haven't changed.)
Outputs the union of these.

Once we have union, we can use them at the nodes of a tree, depicted on p. 16.

The entire tree has size based on n, the potential number of participants.
At least $n2$ leaves.
But the algorithm will only use part of it, thus getting adaptivity.
The union sub-services handle n potential processes each, but again, are adaptive.

Processes first do adaptive $k(k+1)/2$ renaming, and use the resulting name to determine which leaf to start at.
Then climb the tree, participating in unions along the way.
The output from the root is the final output.
Choices begin at the left, so if k is small, only short paths are traversed.

2.8 Significance

Define notions of adaptive (and range-independent) algorithms.
Produces good adaptive algorithms for the renaming and lattice agreement problems.

2.9 Further work

1. Improve the last renaming result by getting an adaptive $2k-1$-renaming algorithm with small polynomial in k time complexity.

   Afek and Merritt have already done this.
   Their algorithm has step complexity $O(k2)$.

   Their algorithm is based on two of the algorithms of this paper: the $6k+1$ algorithm and the shrinking network algorithm.
   First, MA used the $6k+1$ algorithm of this paper to get the namespace down to $6k+1$.
   Then they use a version of the original shrinking network algorithm to get the namespace down from $6k+1$ to $2k+1$.
   This version involves some tricky modifications to the shrinking network algorithm to remove the dependency on a priori bounds.
   Too complicated to present here.

2. What about long-lived renaming? Adaptive versions of this? Attiya-Fouren did some further work on this. I'll mention in a minute.

3. Are versions of these algorithms useful in distributed systems?