Improved Randomized Upper Bounds for Conflict Detectors and Strong Group Renaming
Computer Science 490 Project
Yale University

Michael Kurland
Advisor: James Aspnes

May 8, 2012
Abstract

In this project I give an improved upper bound for the individual step complexity of a conflict detector implemented using multi-writer registers for \( n \) non-anonymous processes with a conflict to detect. I create a conflict detector with expected individual step complexity \( O \left( \min \left( \frac{\log m}{\log \log m}, \log \log n \right) \right) \) when there is a conflict to detect and with guaranteed individual step complexity \( O \left( \min \left( \frac{\log m}{\log \log m}, \log n \right) \right) \) regardless of whether there is a conflict to detect. Here \( m \) is the maximum number of potential values for each process. I also present improved upper bounds for the expected individual step complexity for the strong group renaming problem and for the adaptive strong group renaming problem of \( O \left( \log k \log \log n \right) \) where \( k \) is the number of groups and \( n \) is the number of processes using multi-writer registers and consensus objects (constructed from these registers).
Contents

1 Conflict Detectors ......................................................... 2
  1.1 Introduction ......................................................... 2
     1.1.1 Model ......................................................... 2
     1.1.2 Notation ....................................................... 3
     1.1.3 Adopt-Commit ................................................ 3
  1.2 Strategy for an $O(\log \log n)$ conflict detector when a conflict is present ......................................................... 3
     1.2.1 Incomplete Conflict Detector ......................... 3
     1.2.2 Making a full conflict detector ....................... 8
  1.3 Conclusions ........................................................... 9
     1.3.1 Strength of the adversary .................................. 10

2 Randomized Group Renaming .............................................. 11
  2.1 Introduction .......................................................... 11
     2.1.1 Problem Statement ........................................... 11
     2.1.2 Model .......................................................... 11
  2.2 Renaming ............................................................... 12
     2.2.1 Sorting networks .............................................. 12
     2.2.2 Renaming networks .......................................... 12
  2.3 Non-adaptive group renaming ........................................ 13
  2.4 Adaptive strong group renaming .................................... 14
     2.4.1 An adaptive sorting network ............................. 14
     2.4.2 Adaptive strong group renaming algorithm ............ 17
  2.5 Conclusion ............................................................. 20

3 Bibliography ............................................................... 21
Chapter 1
Conflict Detectors

1.1 Introduction

Conflict detectors are a class of objects that support a single operation, check$(v)$, with input $v$ from a set of $m$ values [9]. This function returns true to indicate a conflict or false to indicate that there are no conflicts. If all check operations have the same input value, then all of these operations must return false. If there are two operations with different input, then at most one input value returns false, with all others returning true [9]. Aspnes and Ellen show how to build these objects from adopt-commit objects and give upper and lower bounds (described below). In this chapter, I propose an algorithm that improves on the upper bound of these objects. Since with anonymous processes the upper and lower bounds match, the improvement depends on taking advantage of identities and it only provides improvement in the special case where there is a conflict to detect.

1.1.1 Model

In building an improved conflict detector, I consider the standard asynchronous shared-memory model, wherein $n$ processes communicate by executing operations on atomic multi-writer multi-reader registers. These registers support a write operation and a read operation which returns the value of the last write.

In the model used, timing is controlled by an oblivious adversary, that specifies a schedule consisting of a sequence of process ids. At each step, the next process in the schedule executes one operation of its choosing. Any coin-flips done by the processes are independent of the adversary’s chosen schedule.
1.1.2 Notation
We use log for base-2 logarithm and \( \ln \) for base-\( e \) logarithm.

1.1.3 Adopt-Commit
An adopt-commit object, also known as a ratifier, is an object that can be used to achieve agreement and consensus [9]. An \( m \)-valued adopt-commit object is defined as an object supporting the operation \( \text{adoptCommit}(u) \) where \( u \) is an input value from a set \( V \) of \( m \) values. There are two potential results of this operation: (commit, \( v \)) or (adopt, \( v \)). Here \( v \in V \), and commit and adopt represent the decision bit that indicates whether a process should immediately commit to \( v \) or simply just adopt it as its preferred value in later rounds of the protocol.

Aspnes and Ellen give a matching upper and lower bound of \( \Theta \left( \min \left( \log m \log \log m, n \right) \right) \) for the space and individual step complexity of a wait-free \( m \)-valued adopt commit object that uses multi-writer registers for \( n \) anonymous processes [9]. This lower bound holds for both randomized and deterministic algorithms. For non-anonymous systems, however, their upper and lower bounds do not match, with a lower bound of \( \Omega \left( \min \left( \log m \log \log m, \sqrt{\log n \log \log n} \right) \right) \) and an upper bound of \( O \left( \min \left( \log m \log \log m, n \right) \right) \). In this section, I describe a new wait-free algorithm to achieve conflict detection with expected individual step complexity of \( O(\log \log n) \) when a conflict is present.

1.2 Strategy for an \( O(\log \log n) \) conflict detector when a conflict is present

1.2.1 Incomplete Conflict Detector
Algorithm [1] is an algorithm that detects a conflict with high probability if there is one but does not guarantee detection. Therefore, if it does not detect a conflict, we cannot be certain that there is no conflict. We will call this an incomplete conflict detector. By combining this algorithm with a second algorithm, we are able to achieve an \( O(\log \log n) \) conflict detector when a conflict is present.

The mechanism for an incomplete conflict detector is as follows. First we reduce the number of participating process ID, using a protocol similar to the sift protocol originally developed in test-and-set in [4] and then expanded with the algorithms for conciliators in [8]. As in [8], a process that sees another process continues with the algorithm rather than dropping out, but adopts the identity of the other process.
\begin{algorithm}
\begin{algorithmic}
\State $\text{myid} \leftarrow \text{pid}$; \hfill // pid is the process's id
\State $\text{value}[\text{pid}] \leftarrow \text{myvalue}$;
\For{$i = 1 \ldots \lceil \log \log n \rceil + \lceil \log \frac{4}{3} (8/\epsilon) \rceil$}
\State $\text{randomBits}[\text{pid}][i] \leftarrow \text{randomBit}(i)$;
\EndFor
\For{$i = 1 \ldots \lceil \log \log n \rceil + \lceil \log \frac{4}{3} (8/\epsilon) \rceil$}
\If{$\text{conflict} = \text{true}$}
\State \text{return true}
\EndIf
\If{$\text{randomBits}[\text{myid}][i] = 1$}
\State $r_i \leftarrow \text{myid}$;
\Else
\If{$r_i \neq \bot$}
\State $\text{myid} \leftarrow r_i$;
\If{$\text{value}[r_i] \neq \text{myvalue}$}
\State $\text{conflict} \leftarrow \text{true}$;
\State \text{return true};
\EndIf
\EndIf
\EndIf
\EndFor
\State \text{return uncertain};
\end{algorithmic}
\end{algorithm}

\textbf{Algorithm 1: Incomplete Conflict Detector}

Prior to entering the first round, each process generates an array of random bits of length $\lceil \log \log n \rceil + \lceil \log \frac{4}{3} (8/\epsilon) \rceil$ and writes its value to a unique location dependent on its process ID to a shared register. Following this, each process goes through a series of $\lceil \log \log n \rceil + \lceil \log \frac{4}{3} (8/\epsilon) \rceil$ rounds. In each round $i$, a process carries out at most three operations. First, it reads the conflict bit to see if some other process has already found a conflict. If a conflict has been reported, it returns true, indicating that there is a conflict. If no conflict has been found, it chooses whether to write or read the register $r_i$, based on the random bit written for round $i$ for its current ID. If the process writes, its ID is retained in the next round. If it reads, its ID is retained if and only if it sees an empty register. Otherwise, there are two possible actions. If it sees the same value as its value but a different ID, it adopts whatever ID it sees.
If it sees a different value, then it turns on the conflict bit \( \square \) and the process returns saying it found a conflict. Since all processes with the same ID in round \( i \) use the same random bit, they all take the same action.

As in the sift algorithm in [8], the probabilities of each event vary from round to round and are tuned to reduce the expected number of IDs as quickly as possible. The details come mainly from lemmas 1, 2, and 3 originally from [8].

**Lemma 1.** Let \( Y_i \) be the number of distinct IDs that survive the first \( i \) rounds of the Algorithm and let \( X_i = Y_i - 1 \) be the number of excess IDs. Then

\[
E[X_{i+1} | X_i] < \min \left\{ p_{i+1} X_i + \frac{1}{p_{i+1}}, (1 - p_{i+1} + p_{i+1}^2) X_i \right\}
\]

**Proof.** We will first show that \( E[X_{i+1} | X_i] < p_{i+1} X_i + \frac{1}{p_{i+1}} \) by obtaining a bound on \( E[Y_{i+1} | Y_i] \) and manipulating it to obtain the bound on \( E[X_{i+1} | X_i] \).

Order the IDs \( id_1, \ldots, id_{Y_i} \) that appear as the current ID of at least one process leaving round \( i \) by the order that a process carrying that ID is first scheduled to write or read \( r_{i+1} \). Note that the assignment of values to processes in round \( i+1 \) is determined by the randomBits for rounds 1 through \( i \) and the schedule selected by the adversary. Both of these are independent of the round-(\( i + 1 \)) randomBits.

For each ID \( id_j \), it survives round \( i + 1 \) under the following conditions:

(a) \( \text{randombits}[id_j][i+1] = 1 \) and so some process writes \( id_j \)

(b) Some process reads \( id_j \)

(c) \( \text{randombits}[id_j][i+1] = 0 \) but some process sees \( \bot \) when it reads

Note that if case (b) occurs, so does case (a) because some process had to write \( id_j \) for some other process to read it. Case (c) occurs if \( \text{randombits}[id_j][i+1] = 0 \) for \( j \) and \( \text{randombits}[id_k][i+1] = 0 \) for all \( k < j \). The probabilities for these events are \( p \) for case (a) and \( (1 - p)^j \) for case (c). Summing over all \( j \) we find that \( E[Y_{i+1} | Y_i] \leq p_{i+1} Y_i + \frac{1}{p_{i+1}^2} - 1 \) where \( \frac{1}{p_{i+1}^2} - 1 = \sum_{j=1}^{\infty} (1 - p)^j \) is an upper bound on the terms from case (b).

By substituting in \( X_i = Y_i - 1 \) and \( Y_i = X_i + 1 \) we get

\[
E[X_{i+1} | X_i] \leq p (X_i + 1) + 1/p - 2
= pX_i + 1/p + p - 2
< pX_i + 1/p
\]

\(^1\text{We assume that the register for the conflict bit is initially set to 0. When it is turned on, it is set to 1.}\)
We now show that $E[X_{i+1}|X_i] < (1 - p_{i+1} + p_{i+1}^2) X_i$. For this bound, we consider separately the cases were the first process $p$ reads or writes. In order to obtain a simple bound, we further assume that all values survive if $p$ reads. More formally we have:

$$E[X_{i+1}|X_i] = (1 - p_{i+1}) E[X_{i+1}|X_i, p \text{ reads}] + E[X_{i+1}|X_i, p \text{ writes}]$$

$$\leq (1 - p_{i+1}) X_i + p_{i+1}^2 X_i$$

$$= (1 - p_{i+1} + p_{i+1}^2) X_i$$

Now we choose the probabilities $p_i$.

We start with $X_0 \leq n - 1$, and find that the first bound in Lemma 1 is minimized by letting $p_1 = 1/\sqrt{X_0}$. This means that $E[X_1] \leq 2\sqrt{X_0}$.

By iterating this procedure in following rounds, we get the recurrence

$$x_0 = n - 1$$

$$p_{i+1} = 1/\sqrt{x_i}$$

$$x_{i+1} = p_i x_i + 1/p_i = 2\sqrt{x_i}$$

The solution to this recurrence is

$$x_i = 2^{2 - 2^{-i+1}} (n - 1)^{2^{-i}} \quad (1.1)$$

$$p_i = 2^{1 - 2^{-i+1}} (n - 1)^{2^{-i}} \quad (1.2)$$

When we use these values of $p_i$ for the first log log $n$ iterations of Algorithm 1 we obtain the following Lemma.

**Lemma 2.** Let $X_i$ be the number of distinct values that survive the first $i$ rounds of the Algorithm using $p_i$ as defined as

$$p_i = 2^{1 - 2^{-1+i}} (n - 1)^{-2^{-i}} \quad (1.3)$$

for $i = 1 \ldots \log \log n$. Let $x_i$ be defined as

$$x_i = 2^{2 - 2^{-i+1}} (n - 1)^{2^{-i}} \quad (1.4)$$

Then $E[X_i] < x_i$ for all $i \in 1 \ldots \log \log n$.
Proof. The proof is done by induction on the round $i$, using Lemma 1 and the recurrence at each step:

$$E[X_{i+1}] = E[E[X_{i+1}|X_i]]$$

$$\leq E[p_{i+1}X_i + 1/p_{i+1}]$$

$$= p_{i+1}E[X_i] 1/p_{i+1}$$

$$\leq p_{i+1}x_i + 1/p_{i+1}$$

$$= 2\sqrt{x_i}$$

$$= x_{i+1}\qed$$

For $i = \lceil \log \log n \rceil$, this gives that

$$x_{\lceil \log \log n \rceil} = 2^{2^{-\lceil \log \log n \rceil + 1}}(n - 1)^{2^{-\lceil \log \log n \rceil}}$$

$$< 4n^{1/\log n}$$

$$= 8$$

For rounds after the first $\lceil \log \log n \rceil$, we switch to $p_i = 1/2$, which minimizes $1 - p_i + p_i^2$ which is the second case of the bound of Lemma 1. This gives us the following Lemma.

**Lemma 3.** Let $X_i$ be the number of distinct values that survive the first $i$ rounds of the Algorithm. Let $p_i$ be defined as above for $i = 1 \ldots \lceil \log \log n \rceil$ and $1/2$ for larger $i$. Let $j > 0$. Then $E[X_{\lceil \log \log n \rceil + j}] \leq 8 \cdot (3/4)^j$

**Proof.** From Lemma 2 we have that:

$$E[X_{\lceil \log \log n \rceil}] \leq 8$$

Each subsequent round multiplies the bound by $(1 - 1/2 + (1/2)^2) = 3/4$. Hence we have that

$$E[X_{\lceil \log \log n \rceil + j}] \leq 8 \cdot (3/4)^j$$

as needed. \qed

Using these lemmas and plugging in the number of iterations from the algorithm we prove the following theorem.

**Theorem 4.** Algorithm 1 implements a conflict detector which detects a conflict when there is one with probability $1 - \epsilon$ using $O(\log \log n + \log(1/\epsilon))$ individual step complexity.
Proof. Note that this proof is based directly on the proof of Theorem 6 in [8]. Termination and validity are obvious from the algorithm, so we concentrate on probabilistic arguments. If there is only one ID (and hence one value) remaining, then any conflict will have been detected and any process that did not start with this value will have detected the conflict.

From Lemma 3, after \(\lceil \log \log n \rceil + \lceil \log_{4/3}(8/\epsilon) \rceil\) rounds, the expected number of excess IDs, \(X\), is at most \(8 \cdot (3/4)^{\log_{4/3}(8/\epsilon)} = 8 \cdot (\epsilon/8) = \epsilon\). So using Markov’s inequality, the probability that \(X\) is nonzero is bounded by \(\epsilon\).

1.2.2 Making a full conflict detector

In order to turn the incomplete conflict detector into a conflict detector, we combine Algorithm 1 with the deterministic algorithm for non-anonymous conflict detection with \(n\) processes and a maximum of \(m\) possible values for the processes described in [9] (which also uses multi-writer multi-reader registers) and has individual step complexity \(O\left(\min\left(\frac{\log m}{\log \log m}, \log n\right)\right)\) regardless of whether there is a conflict. Until a conflict is detected in one of the algorithms or one of the algorithms completes, each process alternates between taking steps in each algorithm. If Algorithm 1 terminates without detecting a conflict, all steps are spent on the algorithm from [9]. The algorithm is guaranteed to terminate in the sum of the number of steps of the two algorithms which is \(O\left(\min\left(\frac{\log m}{\log \log m}, \log n\right)\right)\) regardless of whether there is a conflict.

**Theorem 5.** By alternating Algorithm 1 with a conflict detector of expected individual step complexity of \(O(\log n)\), we can create a conflict detector with expected individual step complexity \(O\left(\min\left(\frac{\log m}{\log \log m}, \log n\right)\right)\) when there is a conflict to detect and with guaranteed individual step complexity \(O\left(\min\left(\frac{\log m}{\log \log m}, \log n\right)\right)\) when there is no conflict to detect.

Proof. Since the conflict detector from [9] is valid and terminates and Algorithm 1 never returns a false conflict and terminates, it is clear that the combined algorithm is valid and terminates. When there is no conflict to detect When there is a conflict, with probability \(1 - \epsilon\) it will be detected by Algorithm 1 in \(O(\log \log n + \log(1/\epsilon))\) individual step complexity. For the remaining proportion of the time when there is a conflict (this proportion is \(\epsilon\), the individual step complexity will be \(O(\log n)\). We will use \(c_j, j \in \mathbb{N}\) to denote the constants. Hence we have that expected individual
step complexity of our full conflict, detector $E[x]$, can be found as follows:

$$E[x] = (1 - \epsilon) \left( c_1 \log \log n + \log \left( \frac{1}{c_2\epsilon} \right) \right) + \epsilon (c_3 \log n)$$

We let

$$\epsilon = \frac{1}{\log n}$$

and substitute in:

$$E[x] = \left( 1 - \frac{1}{\log n} \right) \left( c_1 \log \log n + c_2 \log \left( \frac{1}{\log n} \right) \right) + \frac{1}{\log n} (c_3 \log n)$$

$$\leq \left( c_1 \log \log n + c_2 \log \left( \frac{1}{\log n} \right) \right) + \frac{1}{\log n} (c_3 \log n)$$

$$\leq c_4 (\log \log n + \log (\log n)) + c_3$$

$$= O(\log \log n)$$

Hence the expected individual step complexity for the combined algorithm when there is a conflict is $O(\log \log n)$ as needed.

It is important to note that if the adversary knows the input values, even if there is a conflict, the adversary can guarantee that all processes that have one input value will have step complexity $O(\log n)$ by having these run through the algorithm first without any other processes taking any steps. However, the conflicting processes will all have expected step complexity $O(\log \log n)$ even against this stronger adversary. In general even against the oblivious adversary, the adversary can guarantee that at least one process will take $O(\log n)$ steps even if every process has a different input value. This is because the adversary can run this process to completion before any other process begins.

1.3 Conclusions

Under the assumption of an oblivious adversary, we have shown how to reduce the expected individual step complexity of wait-free consensus from $O(\log n)$ to $O(\log \log n)$ in the standard multi-writer register model when there is a conflict to detect and each process has a unique ID while maintaining the guaranteed $O(\log n)$ individual step complexity even when there is no conflict to detect.
1.3.1 Strength of the adversary

The improved result exploits the limitations of the oblivious adversary in a number of ways. Following the ideas from Aumann [12] and Chandra [14], we pre-determine all random bits that are later shared between may processes. This means that at a minimum we require a content-oblivious adversary, that is an adversary that cannot see the internal states of processes or the contents of registers. The algorithm also chooses between read and write operations probabilistically as in the protocols of Aspnes [8] and Chor, Israeli, and Li [15], thereby requiring a weak adversary that cannot prevent this. It is, however, possible that an adversary stronger than the oblivious adversary could be used while still maintaining the bounds achieved.
Chapter 2

Randomized Group Renaming

2.1 Introduction

The group renaming problem is a generalization of the renaming problem originally described in [10]. In renaming, each process starts with a unique identifier taken from a large namespace. Each process has the goal of selecting a new unique identifier from a smaller name space.

2.1.1 Problem Statement

The group renaming problem is described in [2]. In group renaming there are \( n \) processors, divided into \( m \) groups. Each processor has a group name taken from a large namespace \([M] = \{1, \ldots, M\}\) in addition to a unique process name taken from \([N]\). The objective of each processor is to choose a new group name from \([P]\), where \( P < M \). There are two conditions that the processes must satisfy: consistency, which means that all processes that start with the same group name in \([M]\) must have the same group name in \([P]\), and uniqueness, meaning that any two processes with different group names in \([M]\) must select different group names in \([P]\). In adaptive strong group renaming, the size of \([P]\) is exactly equal to \( m \) but the size of the initial namespace \([M]\) is unknown and may be arbitrarily large [5].

2.1.2 Model

In solving group renaming, I consider the standard asynchronous memory model previously described in Section 1.1.1. We again assume atomic multi-writer multi-
reader registers and an oblivious adversary. Furthermore, we assume that any coin-
flips done by the processes are independent of the adversary’s chosen schedule.

2.2 Renaming

In this section, I describe the use of consensus objects at each node of a renaming
network to build a group renaming network that routes each group of processes
through a sequence of consensus objects to a unique output wire. I adopt two
algorithms from the renaming problem to solve group renaming and strong group
renaming. I describe the background information here.

2.2.1 Sorting networks

A sorting network a parallel sorting algorithm that proceeds in synchronous rounds.
In each round, the elements of an array at certain fixed positions are paired off and
swapped if they are out of order. The choice of which positions to compare at each
step of the sorting network is static and does not depend on the outcome of previous
comparisons. Additionally, the only effect of a comparison is possibly swapping the
two values that were compared.

The depth of a sorting network is the maximum number of comparators on any
path from input to output. The width of a sorting network is the number of input
wires which is equivalently the number of values that can be sorted by the network.
For input of size $n$, it has been proven that a sorting network of depth $O(\log n)$ exists
[3], but no explicit construction of such a network is known at this time [7]. Explicit
construction of sorting networks with width $n$ and depth $O(\log^2 n)$ are known [13].

2.2.2 Renaming networks

To turn a sorting network into a renaming network, we replace the comparators
of the sorting network with test-and-set bits, and allow processes to walk through
the network asynchronously. Each process starts on a separate input wire and the
process that wins the test-and-set takes the lower wire. Starting with a network of
width $N$ where $N$ is the size of the namespace $|N|$, the expected cost for a process
to acquire a name is then $O(\log N)$ by using an AKS sorting network. Alistarh et
al. show how to make this adaptive, giving an expected cost of $O(\log k)$ where $k$ is
the number of processes [5].
2.3 Non-adaptive group renaming

To solve group renaming I modify the renaming network described in Section 2.2.2. At each node of the renaming network, we replace the test-and-set with a consensus object. We will use the consensus objects derived from adopt-commit objects described by Aspnes in [8]. For the non-anonymous processes we have that the expected individual step complexity for consensus is $O(\log \log n)$ where $n$ is the number of processes.

Algorithm 2 gives an implementation of group renaming with expected individual step complexity will be $O(\log m \log \log n)$ and the total expected step complexity will be $O(n \log m \log \log n)$.

```plaintext
1 // consensus(node) returns the value that is the consensus value achieved by the consensus object for node
2 node ← network.inputNodes[groupID];
3 for i = 1 ... \ceil{\log m} do
4     if groupID = consensus(node) then
5         node ← node.down
6     else
7         node ← node.up
8     endif
9 end
10 newid ← node.height;
11 // This height is the group id. Lowest node has 1, and increments as go up
12 return newid
```

Algorithm 2: Non-adaptive group renaming

In Algorithm 2 each process maps to an input wire in the sorting network which is defined based on its group name. If the size of the initial namespace of group names $[M]$ is size $m$ then a sorting network of width $m$ with depth $\ceil{\log m}$ will be used. At each level, a process enters the consensus object at that node of the sorting network. If the consensus value achieved is the same value as that held by a process, then it will take the lower wire in the sorting network. If the consensus value is not the same value as that held by a process, it will take the higher wire in the sorting network.

**Theorem 6.** Algorithm 2 implements strict group renaming with individual step complexity of $O(\log m \log \log n)$ and with total step complexity of $O(n \log m \log \log n)$.
Proof. All processes in a group start together and each group starts at a unique location based on the group ID. At each node in the sorting network, at most two groups reach the node. A single consensus value is achieved at each node. If only one group reaches a node, then its value will be chosen and it will proceed down. If two groups reach a node, then the group that wins consensus will proceed down and the other group will proceed up. Hence all processes that start together will end together. Also, since a sorting network will sort processes from lowest value to highest value and any time a group wins consensus it has the lower value, if there are \( k \) groups, the groups will end on the \( k \) lowest output wires and be given names \( 1, \ldots, k \).

Since consensus can be achieved with expected step complexity of \( O(\log \log n) \), each process is expected to take \( O(\log \log n) \) individual steps at each level of the sorting network. There are \( \lceil \log m \rceil \) levels of the network. Therefore, Algorithm 2 achieves strong consensus with expected individual step complexity of \( O(\log m \log \log n) \) and expected total step complexity \( O(n \log m \log \log n) \).

In many articles including [2] and [1] written about the group renaming problem, the problem is restricted such that no group has more than \( g \) members.

Corollary 7. If group size is known to be no larger than \( g \) then the Algorithm 2 achieves strong consensus with expected individual step complexity of \( O(\log m \log \log g) \) and expected total step complexity \( O(n \log m \log \log g) \).

Proof. The proof is the same as the proof for Theorem 6 except that rather than as many as \( n \) processes entering each consensus protocol, at most \( 2g \) processes enter each consensus protocol. Hence the expected individual step complexity at each level of the network is \( O(\log \log g) \), expected individual step complexity is \( O(\log m \log \log g) \) and expected total step complexity is \( O(n \log m \log \log g) \).

2.4 Adaptive strong group renaming

In this section, I present an algorithm for adaptive strong group renaming based on an adaptive sorting network construction used for adaptive strong renaming in [5].

2.4.1 An adaptive sorting network

Here I present the recursive construction of a sorting network of arbitrary size also presented in [5]. This construction guarantees the properties of a sorting network whenever limited to a finite number of input and output ports [5]. The sorting
network is adaptive in that any value entering on wire \( x \) and leaving on wire \( y \) goes through at most \( O(\max(x,y)) \) consensus objects.

The key observation is that we can extend a small sorting network \( B \) to a larger sorting network by putting it in between two much larger sorting networks, \( A \) and \( C \). The resulting sorting network is non-uniform; different paths through the network have different lengths. The lowest part of the sorting network has the same width as \( B \).

We will now formally describe how to combine three sorting networks into a single larger sorting network. Suppose we have sorting networks \( A \) and \( C \) with width \( m \) and \( B \) with width \( k \). Label the inputs of \( A \) as \( A_1, A_2, \ldots, A_m \) and the outputs as \( A'_1, A'_2, \ldots, A'_m \), where \( i < j \implies A'_i \leq A'_j \) and \( i < j \implies A'_i < A'_j \) if all inputs are different. Label the input and output wires for \( B \) and \( C \) similarly. Fix \( \ell \leq k/2 \) and construct a new sorting network \( ABC \) with inputs \( B_1, B_2, \ldots, B_\ell, A_1, \ldots, A_m \) and outputs \( B'_1, B'_2, \ldots, B'_\ell, A'_1, A'_2, \ldots, A'_m \). Insert \( B \) between \( A \) and \( C \) by connecting outputs \( A'_1, \ldots, A'_{\ell-1} \) to inputs \( B_{\ell+1}, \ldots, B_k \) and outputs \( B'_{\ell+1}, \ldots, B'_k \) to inputs \( C'_1, \ldots, C'_{k-\ell} \). Then the remaining outputs of \( A \) are wired to the corresponding inputs of \( C \). More formally, outputs \( A'_{k-\ell+1}, \ldots, A'_m \) are connected to inputs \( C_{k-\ell+1}, \ldots, C_m \). We now show that the resulting construction is a sorting network by following the proof of Alistarh et al. [5].

**Lemma 8.** The network \( ABC \) constructed as described above is a sorting network.

**Proof.** The proof depends on the Zero-One Principle which states that when a network sorts all input sequences of zeros and ones correctly, it correctly sorts all input sequences. [16]

Given a particular 0-1 input sequence, let \( z_B \) and \( z_A \) be the number of zeroes in the input that are sent to inputs \( B_1 \ldots B_\ell \) and \( A_1 \ldots A_m \) respectively. Since \( A \) sorts all of its incoming zeros to the lowest of its outputs, \( B \) gets a total \( z_B + \min(k-\ell, z_A) \) zeros on its input wires. \( B \) sorts those zeroes to outputs \( B'_1 \ldots B'_{z_B+\min(k-\ell,z_A)} \). An additional \( z_A - \min(k-\ell, z_A) \) zeros continue directly from \( A \) to \( C \). We consider two cases that depend on the value of the min:

**Case 1:** \( z_A \leq k-\ell \). In this case, \( B \) gets \( z_B + z_A \) zeros, which is all of them. Since it sorts them to its lowest outputs, the network puts the zeros on outputs \( B'_1 \ldots B'_\ell \) as it should. The zeros that reach outputs \( B'_{\ell+1} \) and above are not moved by \( C \). Therefore the sorting network works on 0-1 input sequences correctly in this case.

**Case 2:** \( z_A > k-\ell \). In this case \( B \) gets \( z_B + k-\ell \) zeros while the remaining \( z_A - (k-\ell) \) zeros continue directly from \( A \) to \( C \). Because \( \ell \leq k/2 \) we have that \( z_B + k-\ell \geq k/2 \geq \ell \). Therefore \( B \) sends \( \ell \) zeros out to its direct outputs \( B'_1 \ldots B'_\ell \). All remaining values are fed into \( C \), which sorts the remaining zeros into the next
z_A + z_B − ℓ positions as they should be. Therefore the sorting network works on 0-1 input sequences in this case.

Since the network works on all 0-1 input sequences, by the Zero-One Principle, the network ABC works on all sequences correctly and is a sorting network.

In constructing bounds for our adaptive strong group renaming algorithm in Section 2.4.2 it will be important to limit which parts of the network particular values traverse. The key tool to do so is the following Lemma whose proof follows immediately from Lemma 8.

**Lemma 9.** If an value v supplied to one of the inputs B_1 through B_ℓ in the network ABC, and is one of the ℓ smallest values supplied on all inputs, then v never leaves ABC.

Now we will show how to recursively construct a large sorting network with polylog N depth when truncated to the first N positions. We assume that we are using a construction of a sorting network that requires at most \( \log^c n \) depth to sort n values, where a and c are constants. For the AKS sorting networks, we have \( c = 1 \) [3] while for constructible networks we have \( c = 2 \) [16].

We start with a sorting network \( S_0 \) of width 2. In general, we will let \( w_k \) be the width of \( S_k \). Hence we have that \( w_0 = 2 \). We also use \( d_k \) to denote the depth of \( S_k \). Recall that the depth is the number of comparators on the longest path through the network.

Given \( S_k \), we construct \( S_{k+1} \) as follows. We take two sorting networks \( A_{k+1} \) and \( C_{k+1} \) with width \( w_k^2 - w_k/2 \) and attach them to the top half of \( S_k \) as in Lemma 8, setting \( ℓ = w_k/2 \). Note that:

\[
\begin{align*}
w_{k+1} &= w_k^2 \\
d_{k+1} &= 2a \log^c \left( w_k^2 - w_k/2 \right) + d_k \\
&\leq 4a \log^c w_k + d_k
\end{align*}
\]

Solving these recurrences gives that

\[
\begin{align*}
w_k &= 2^{2^k} \\
d_k &= \sum_{i=0}^{k} 2^{c(i+2)}a = O \left( 2^{ck} \right)
\end{align*}
\]

If we set \( N = 2^{2^k} \), then \( k = \log \log N \) and \( d_k = O \left( 2^{c \log \log N} \right) = O \left( \log^c N \right) \) thereby giving us polylogarithmic depth for a network with \( N \) line, and a total number of comparators of \( O \left( N \log^c N \right) \).

Indeed we can actually make a stronger statement, which taken from [5].
Theorem 10. Each of the networks $S_k$ constructed above is a sorting network, with the property that any value that enters on the $n^{th}$ input and leaves on the $m^{th}$ output transverses $O\left(\log^c\max(n,m)\right)$ comparators.

Proof. $S_k$ is a sorting network by induction on $k$ using Lemma 8. To show that any value that enters on the $n^{th}$ input and leaves on the $m^{th}$ output transverses $O\left(\log^c\max(n,m)\right)$ comparators, let $S_{k'}$ be the smallest stage in the construction of $S_k$ to which input $n$ and output $m$ are directly connected. Then:

\[
\begin{align*}
    w_{k'-1}/2 &< \max(n,m) \leq w_{k'}/2 \\
    2^{2^{k'-1}} &< 2\max(n,m) \leq 2^{2^{k'}} \\
    k'-1 &< \log \log \max(n,m) \leq k' \\
    \implies k' &\leq \lceil \log \log \max(n,m) \rceil
\end{align*}
\]

By Lemma 9, the given value stays in $S_{k'}$ which means that it traverses at most $d_{k'}$ comparators where

\[
d_{k'} = O\left(2^{ck'}\right) = O\left(2^c\lceil \log \log \max(n,m) \rceil\right) = O\left(\log^c \max(n,m)\right)
\]

2.4.2 Adaptive strong group renaming algorithm

We now show how to use the adaptive sorting network construction to solve strong adaptive renaming when the size of the initial group namespace, $[M]$, is unknown, and may be arbitrarily large.

Description

Our algorithm is composed of two stages and is an adaption of the strong adaptive renaming algorithm of Alistarh et al. [5]. In the first stage, each process obtains a temporary group name in a namespace of size polynomial in $k$ where $k$ is the number of non-empty groups, with high probability. This stage is referred to as $\text{GroupTempName}$. We allocate a binary tree of consensus objects of unbounded height that act as randomized group splitters. We will call these objects $\text{consensus splitters}$. This structure for randomized splitters was described in [11]. Each of these consensus objects is a component such that one group name may “win” the splitter by having its group name decided as the consensus value. The losing processes go left or right with probability $1/2$. To ensure that groups stay together, each group
will have a shared set of random bits and if the random bit for this depth matches
the random bit for the winning group, the group goes right, otherwise it goes left.
Achieving these shared random bits can be achieved through a consensus protocol
and can therefore be achieved in \( O(\log \log g) \) where \( g \) is the size of the largest group
[8]. Each process starts the protocol at the root consensus splitter in the tree. If
the process's group does not win the current root consensus splitter, the entire goes
either left or right, each with probability \( \frac{1}{2} \) until the group manages to acquire a
consensus splitter. Since each consensus splitter reached must be won, the process
will stop at height at most \( k \). Once a process (and hence its entire group) stops,
the process adopts a temporary name corresponding to the index of the splitter in a
breadth-first search labeling of the tree nodes.

In the second stage of the algorithm, we use the group renaming network described
in Section 2.2.2 and in particular using the adaptive group sorting network of Section
2.4.1. Let \( R \) be the resulting group renaming network. Each process in a group uses
the temporary group name it has acquired in the first stage as the index of its input
write to the renaming network \( R \). The process then executes the group renaming
network \( R \) starting at the given input wire and returns the index of its output port
as its group name.

**Wait-freedom**

It is important to note that technically this algorithm may not be wait-free if \( k \) is
unbounded. If the number of group names \( k \) participating in an execution is \textit{infinite},
then it possible that a group fails to acquire a temporary group name during the first
stage or it continually fails to reach an output wire by always losing the consensus
objects it participates in. Therefore, for analysis of the algorithm, we assume that \( k \)
is finite. The bounds on individual step complexity presented depend on \( k \).

**Analysis**

**Theorem 11.** The GroupTempName algorithm has the following properties:

1) Given \( k \) participating group names, GroupTempName assigns names between
1 and \( k^c \) with probability \( 1 - \frac{1}{k^{c-1}} \) where \( c > 1 \) is a constant.

2) GroupTempName has individual step complexity \( O(\log k \log \log n) \) with high
probability in \( k \) and \( n \) where \( n \) is the number of processes.

**Proof.** We consider the first part of the theorem first. Consider a tree with depth
\( c \log k \). This tree contains names \( 1 \ldots k^c \) as \( 2^{\log k} - 1 = k^c - 1 \). The probability that
2 particular group names have taken the exact same direction at each group splitter is bounded above by

\[
\left(\frac{1}{2}\right)^{c\log k} = (2^{-1})^{c\log k} = 2^{\log k^{-c}} = k^{-c}
\]

Parts of this proof are modeled off of [6]. The probability that any 2 group names have taken the exact same direction at each splitter is less than or equal to \(k\) times the probability that 2 particular group names have taken the exact same direction at each group splitter by the union bound. Therefore, then probability that any 2 group names have taken the exact same direction at each group splitter is less than or equal to \(k \times k^{-c} = k^{1-c}\). Therefore the probability that no 2 group names have taken the exact same path is greater than or equal to

\[
1 - k^{1-c} = 1 - 1/k^{c-1}
\]
as needed.

To show that the second part of the theorem is true, we assume that either a group is named within depth \(c\log k\) of the tree; otherwise some process must go to a depth of \(k\) to be named. This is an upper bound on the expected number of group splitters the worst group will encounter in the algorithm algorithm. Hence we have that

\[
E[\text{worst individual group splitters encountered}] \leq (c\log k) + (1/k^{c-1}) k
\]

\[
= c\log k + k^{2-c}
\]

By letting \(c = 2\) we have that

\[
E[\text{worst individual group splitters encountered}] \leq c\log k + k^{2-c}
\]

\[
= 2\log k + k^{2-2}
\]

\[
= 2\log k + 1
\]

Since each splitter is simply a consensus protocol with fixed additional steps and consensus can be performed in \(O(\log \log n)\) we have that GroupTempName has expected individual step complexity \(O(\log k \log \log n)\) as needed.

Using Theorem 11 we prove the following theorem adapted from [5].

**Theorem 12.** For any finite \(k > 0\), the adaptive group renaming network construction on the AKS sorting network solves adaptive strong group renaming for \(k\) group names and \(n\) processes. The expected individual step complexity is \(O(\log k \log \log n)\).

**Proof.** We first prove that the resulting construction solves adaptive strong group renaming for any \(k > 0\). First we know that the temporary names obtained in the first
stage are between 1 and \(k^c\) with high probability for a constant \(c \geq 1\). Therefore, we will assume that during the current execution, each group enters an input wire of the group renaming network between 1 and \(k^c\). We truncate the renaming network after the first \(k^c\) input wires. By Theorem 10 we obtain that the original comparison network truncated after the first \(k^c\) input wires is indeed a sorting network. The second stage of the construction implements adaptive strong group renaming for at most \(k^c\) processes. The first claim follows.

For the complexity bound, we have from Theorem 11 that any process takes \(O(\log k \log \log n)\) steps during the first stage with high probability. From Theorem 10 we have that the number of consensus protocols a process competes in during an execution of the group renaming network is \(O(\log \max (\ell, m))\), where \(\ell\) is the number of the input wire for the process’s group and \(m\) is the number of the output wire for the process and its group. We note that \(\ell \leq k^c\) with high probability and \(m \leq k\) by the adaptive tight property of the group renaming network.

Therefore, a process competes in \(O(\log k)\) group splitters in the second stage, each with step complexity \(O(\log \log n)\). Hence we have that the expected step complexity for a process is \(O(\log k \log \log n)\).

\[\Box\]

### 2.5 Conclusion

Under the assumption of an oblivious adversary, we have shown how to solve the adaptive strong group renaming problem with expected individual step complexity of \(O(\log k \log \log n)\) in the standard multi-writer register model. We make no assumption that the processes know the number of groups. If the processes have knowledge of the groups (either their size or their number), this could potentially be used to improve the bound on the expected individual step complexity of the algorithm. This is a potential area to expand on this work.
Chapter 3

Bibliography


