Group Renaming

M.Sc. degree in the School of Computer Science, Tel-Aviv University

by

Irit Levy

The research work for this thesis has been carried out at Tel-Aviv University, Israel.
under the supervision of Prof. Yehuda Afek

December 2011

\[1\] This work is an expansion of [1]
Abstract

This thesis addresses the *group renaming* task, which is a natural generalization of the *renaming* task. An instance of this task consists of *n* processors, partitioned into *m* groups, each of at most *g* processors. Each processor knows the name of its group, which is in \{1, \ldots, M\}. The task of each processor is to choose a new name for its group such that processors from different groups choose different new names from \{1, \ldots, \ell\}, where \(\ell < M\). We consider two variants of the problem: a *tight* variant, in which processors of the same group must choose the same new group name, and a *loose* variant, in which processors from the same group may choose different names. Our findings can be briefly summarized as follows:

1. We present an algorithm that solves the tight variant of the problem with \(\ell = 2m - 1\) in a system consisting of \(g\)-consensus objects and atomic read/write registers. In addition, we prove that it is impossible to solve this problem in a system having only \((g - 1)\)-consensus objects and atomic read/write registers.

2. We devise a \(\{n, m, g\}\)-independent algorithm for the loose variant of the problem that only uses atomic read/write registers, and has \(\ell = 4n - 3\). The algorithm also guarantees that the number of different new group names chosen by processors from the same group is at most \(\min\{g, 2m, 3\sqrt{n} - 1\}\). Furthermore, we consider the special case when the groups are uniform in size and show that our algorithm is adaptive to have \(\ell = 2m(m + 1) - 3\), when \(m < \sqrt{n}\), and \(\ell = 4n - 3\), otherwise.
Acknowledgements

This thesis is a joint work with Professor Yehuda Afek.

Parts of this thesis are also a joint work with Iftah Gamzu, Opher Lieber, Michael Merritt and Gadi Taubenfeld.

I would like to thank Professor Afek for his guidance, assistance and dedication throughout the work on this thesis.
## Contents

1 Introduction ......................................................... 5

2 Model and Problem Definitions ................................. 8

3 Tight Group Renaming ........................................... 11
    3.1 An upper bound ............................................. 11
    3.2 An impossibility result .................................... 16

4 Loose Group Renaming ........................................... 20
    4.1 The non-uniform case ....................................... 20
    4.2 The uniform case ........................................... 30

5 Discussion ....................................................... 32
Chapter 1

Introduction

The goal of a distributed algorithm is to utilize the communication between processors in a distributed network in order to achieve a common goal. There are two major communication models for executing a distributed computation, message passing and shared memory. Systems that communicate via shared memory objects may differ in the levels of atomicity they support. We assume that any system operating under the shared memory model supports atomic read/write operations in order to exchange information. The wait-free renaming task is a fundamental distributed problem under the atomic read/write model. Each processor participating in the renaming task starts executing with a unique identifier taken from a large domain. The goal of each processor is to select a new unique identifier from a smaller range. Such an identifier can be used, for example, to mark a memory slot in which the processor may publish information in its possession. A tight upper bound for the wait-free renaming task was first presented by Attiya et al [4]. Assuming that the system consists of $n$ processors, Attiya’s algorithm chooses $n$ new unique identifiers from the range $\{1, \ldots, 2n - 1\}$.

This thesis investigates the group renaming task which is a generalization of the renaming task for groups of processors. Groups of processors that participate in the group renaming task may hold some information which they would like to publish, preferably using a common memory slot for each group. An additional motivation for studying the group version of the problem is to further our understanding about the inherent difficulties in solving tasks with respect to groups [11].
More formally, an instance of the group renaming task consists of \( n \) processors partitioned into \( m \) groups, each of which consists of at most \( g \) processors. Each processor has a group name taken from some large name space \([M] = \{1, \ldots, M\}\), representing the group that the processor affiliates with. In addition, every processor has a unique identifier taken from \([N]\). The objective of each processor is to choose a new group name from some name space \([\ell]\), where \( \ell < M \). The collection of new group names selected by the processors must satisfy the uniqueness property meaning that any two processors from different groups choose distinct new group names. We consider two variants of the problem:

- a tight variant, in which in addition to satisfying the uniqueness property, processors from the same group must choose the same new group name (this requirement is called the consistency property).
- a loose variant, in which processors from the same group may choose different names rather than a single one, as long as the uniqueness property is maintained.

This thesis presents a wait-free algorithm that solves the tight variant of the group renaming problem with \( \ell = 2m - 1 \) in a system equipped with \( g \)-consensus objects and atomic read/write registers. This algorithm extends the upper bound result of Attiya et al. [4] for \( g = 1 \). On the lower bound side, we show that there is no wait-free implementation of tight group renaming in a system equipped with \((g - 1)\)-consensus objects and atomic read/write registers. In particular, this result implies that there is no wait-free implementation of tight group renaming using only atomic read/write registers for \( g \geq 2 \).

We then restrict our attention to shared memory systems which support only atomic read/write registers and study the loose variant of the group renaming problem. An instance of the loose group renaming task consists of \( n \) processors, partitioned into \( m \) groups, each group of at most \( g \) processors. Each processor knows the name of its group, which is in \([1, \ldots, M]\). The task of each processor is to choose a new name for its group such that processors from different groups choose different new names from \([1, \ldots, \ell]\), where \( \ell < M \), while processors from the same group may choose the same name.

We present an algorithm for the loose group renaming task which is \( \{n, m, g\} \)-independent, in other
words the algorithm does not know $n$, $m$ or $g$ a-priori and is independent of these parameters. It runs in the standard atomic read and write shared memory model, has $\ell = 4n - 3$ in the worst case and guarantees that the number of different new group names chosen by processors from the same group is bounded by $\min\{g, 2m, 3\sqrt{n} - 1\}$. It seems worthy to note that the algorithm is built around a filtering technique that overcomes "hard" scenarios in which both the size of the maximal group and the number of groups are large, i.e., $g = \Omega(n)$ and $m = \Omega(n)$. Essentially, such scenario arises when there are $\Omega(n)$ groups containing only few members and few groups containing $\Omega(n)$ members.

Furthermore, we consider the special case when the groups are uniform in size and show that our algorithm is adaptive to have $\ell = 2m(m + 1) - 3$, when $m < \sqrt{n}$, and $\ell = 4n - 3$, otherwise. This last result settles, to some extent, an open question posed by Gafni [11].
Chapter 2

Model and Problem Definitions

Our model of computation consists of an asynchronous collection of $n$ processors communicating via shared objects. Each object has a type which defines the set of operations that the object supports. Each object also has sequential specification that specifies how the object behaves when these operations are applied sequentially.

Various systems differ in the level of atomicity that is supported. Atomic (or indivisible) operations are defined as operations that their execution is not interfered by other concurrent activities. This definition of atomicity is too restrictive, and it is safe to relax it by assuming that processors can try to access the object at the same time, however, although operations of concurrent processors may overlap, each operation should appear to take effect instantaneously. In particular, operations that do not overlap should take effect in their “real-time” order. This type of correctness requirement is called linearizability \[14\].

We will always assume that the system supports atomic registers, which are shared objects that support atomic reads and writes operations. In addition, the system may also support forms of atomicity which are stronger than atomic reads and writes. One specific atomic object that will play an important role in our investigation is the consensus object. A consensus object $o$ supports one operation: $o.propose(v)$, satisfying:

1. Agreement. In any run, the $o.propose()$ operation returns the same value, called the consensus
value, to every processor that invokes it.

2. Validity. In any run, if the consensus value is \( v \), then some processor invoked \( o.propose(v) \).

The notation \( g \)-consensus is used to denote a consensus object for \( g \) processors.

An object is \textit{wait-free} if it guarantees that \textit{every} processor may always complete its pending operation in a finite number of its own steps regardless of the execution speed of other processors (does not admit starvation). Similarly, an implementation or an algorithm is wait-free, if every processor makes a decision within a finite number of its own steps. We will focus only on wait-free objects, implementations or algorithms. Next, we define two notions for measuring the relative computational power of shared objects.

- The \textit{consensus number} of an object of type \( o \), is the largest \( n \) for which it is possible to implement an \( n \)-consensus object in a wait-free manner, using any number of objects of type \( o \) and any number of atomic registers. If no largest \( n \) exists, the consensus number of \( o \) is infinite.

- The \textit{consensus hierarchy} (also called the wait-free hierarchy) is an infinite hierarchy of objects such that the objects at level \( i \) of the hierarchy are exactly those objects with consensus number \( i \).

It has been shown in [13], that in the consensus hierarchy, for any positive \( i \), in a system with \( i \) processors: (1) no object at level less than \( i \) together with atomic registers can implement any object at level \( i \); and (2) each object at level \( i \) together with atomic registers can implement any object at level \( i \) or at a lower level, in a system with \( i \) processors. Classifying objects by their consensus numbers is a powerful technique for understanding the relative power of shared objects.

The solutions for the tight group renaming problem, presented in this work, relay on these notions. Therefore, we will assume that \( m \), the number of groups, is greater or equal to 2.

\textbf{Evolution of the group renaming problem}  The renaming problem was first solved for message-passing systems [4], and only then for shared memory systems using atomic registers [7].
Both these papers present one-shot algorithms (i.e., solutions that can be used only once). The first long-lived renaming algorithm, the $\ell$-assignment algorithm, was presented in [9]. The $\ell$-assignment algorithm can be used as an optimal long-lived $(2n - 1)$-renaming algorithm with exponential step complexity. Improving the runtime, [5] was the first wait-free long lived renaming algorithm that adapts to the number of simultaneously active processes using atomic registers. This algorithm has a step complexity of $O(k \log k)$ where $k$ is the number of active processes. Several of the many papers on renaming using atomic registers are [2, 3, 12, 15, 16]. Other references are mentioned later on.

The group solvability problem, which investigates the solvability of tasks by groups of processors, was first introduced and investigated in [11]. In that paper, the solvability of the renaming task by groups, which we investigate in this thesis, was first discussed.
Chapter 3

Tight Group Renaming

3.1 An upper bound

In what follows, we present a wait-free algorithm that solves tight group renaming using $g$-consensus objects and atomic registers. Essentially, we prove the following theorem.

**Theorem 3.1.1.** For any $g \geq 1$, there is a wait-free implementation of tight group renaming with $\ell = 2m - 1$ in a system consisting of $g$-consensus objects and atomic registers.

**Corollary 3.1.2.** The consensus number of tight group renaming is at most $g$. 
Algorithm 1 Tight group renaming algorithm: code for processor $i \in [N]$. 

In shared memory:
- $SS[1, \ldots, N]$ array of swmr registers, initially $\perp$.
- $HIS[1, \ldots, N][1, \ldots, I][1, \ldots, N]$ array of swmr registers, initially $\perp$.
- $CON[1, \ldots, M][1, \ldots, I]$ array of $g$-consensus objects.

1: $p \leftarrow 1$
2: $k \leftarrow 1$
3: while true
4: \begin{itemize}
  \item [] SS[$i$] $\leftarrow \langle \text{GID}_i, p, k \rangle$
  \item [] HIS[$i$][$k$][1, \ldots, N] $\leftarrow$ Snapshot(SS)
\end{itemize}
5: 
\begin{itemize}
  \item [] Agree on $w$, the winner of group $\text{GID}_i$ in iteration $k$, and import its snapshot:
  \item [] $w \leftarrow \text{CON}[\text{GID}_i][k].\text{Compete}(i)$
  \item [] HIS[$w$][$k$][1, \ldots, N]
\end{itemize}
6: \begin{itemize}
  \item [] $P = \{p_j : j \in [N] \text{ has } \text{GID}_j \neq \text{GID}_w \text{ and } k_j = \max_{q \in [N]}\{k_q : \text{GID}_q = \text{GID}_j\}\}$
\end{itemize}
7: if $p_w \in P$
8: \begin{itemize}
  \item [] $r \leftarrow$ the rank of $\text{GID}_w$ in $\{\text{GID}_j \neq \perp : j \in [N]\}$
  \item [] $p \leftarrow$ the $r$-th integer not in $P$
\end{itemize}
9: else return $p_w$
10: \end{itemize}
11: $k \leftarrow k + 1$
12: end while

Our implementation, i.e., Algorithm 1, is inspired by the renaming algorithm of Attiya et al. [4], which achieves an optimal new name space size of $2n-1$. In this renaming algorithm, each processor iteratively picks some name and suggests it as its new name until an agreement on the collection of new names is reached. The communication between the processors is done using an atomic snapshot object. Our algorithm deviates from this scheme by adding an agreement step between processors of the same group, implemented using $g$-consensus objects. Intuitively, this agreement step ensures that all the processors of any group will follow the decisions made by the “fastest” processor in the group. Consequently, the selection of the new group names can be determined between the representatives of the groups, i.e., the “fastest” processors. This enables us to obtain the claimed new names space size of $2m-1$. It is worthy to note that the “fastest” processor of some group may change over time, and hence our agreement step implements a “follow the (current) group leader” strategy. We believe that this concept may be of independent interest. Note that the group name of processor $i$ is designated by $\text{GID}_i$, and the overall number of iterations executed is marked by $I$.

We now turn to establish Theorem 3.1.1. Essentially, this is achieved by demonstrating that
Algorithm 1 maintains the consistency and uniqueness properties (Lemmas 3.1.4 and 3.1.5), that it has $\ell = 2m - 1$ (Lemma 3.1.6), and that it terminates after a finite number of steps (Lemma 3.1.7).

Let us denote the value of $p$ written to the snapshot array (see line 4) in some iteration as the *proposal value* of the underlying processor in that iteration.

**Lemma 3.1.3.** The proposal values of processors from the same group are identical in any iteration.

*Proof.* Consider some group. One can easily verify that the processors of that group, and in fact all the processors, have an identical proposal value of 1 in the first iteration. Thus, let us consider some iteration $k > 1$ and prove that all these processors have an identical proposal value. Essentially, this is done by claiming that all the processors update their value of $p$ in the preceding iteration in an identical manner. For this purpose, notice that all the processors compete for the same $g$-consensus object in that iteration, and then import the same snapshot of the processor that won this consensus (see lines 6–7). Consequently, they execute the code in lines 8–13 in an identical manner. In particular, this guarantees that the update of $p$ in line 11 is done exactly alike. \(\square\)

**Lemma 3.1.4.** All the processors of the same group choose an identical new group name.

*Proof.* The proof of this lemma follows the same line of argumentation presented in the proof of Lemma 3.1.3. Again, the key observation is that in each iteration, all the processors of some group compete for the same $g$-consensus object, and then import the same snapshot. Since the decisions made by the processors in lines 8–13 are solely based on this snapshot, it follows that they are identical. In particular, this ensures that once a processor chooses a new group name, all the other processors will follow its lead and choose the same name. \(\square\)

**Lemma 3.1.5.** No two processors of different groups choose the same new group name.

*Proof.* Recall that we know, by Lemma 3.1.4, that all the processors of the same group choose an identical new group name. Hence, it is sufficient that we prove that no two groups select the same new name. Assume by way of contradiction that this is not the case, namely, there are two distinct groups $G$ and $G'$ that select the same new group name $p^*$. Let $k$ and $k'$ be the iteration numbers in which the decisions on the new names of $G$ and $G'$ are done, and let $w \in G$ and $w' \in G'$ be
the corresponding processors that won the $g$-consensus objects in that iterations. Now, consider the snapshot $(\langle \text{GID}_1, p_1, k_1 \rangle, \ldots, \langle \text{GID}_N, p_N, k_N \rangle)$, taken by $w$ in its $k$-th iteration. One can easily validate that $p_w = p^*$ since $w$ writes its proposed value before taking a snapshot. Similarly, it is clear that $p^\prime_{w^\prime} = p^*$ in the snapshot $(\langle \text{GID}^{\prime}_1, p^{\prime}_1, k^{\prime}_1 \rangle, \ldots, \langle \text{GID}^{\prime}_N, p^{\prime}_N, k^{\prime}_N \rangle)$, taken by $w^\prime$ in its $k^\prime$-th iteration. By the linearizability property of the atomic snapshot object and without loss of generality, we may assume thatsnapshot of $w$ was taken before the snapshot of $w^\prime$. Consequently, $w^\prime$ must have captured the proposal value of $w$ in its snapshot, i.e., $p^\prime_{w^\prime} = p^*$. This implies that $p^*$ appeared in the set $P$ of $w^\prime$. However, this violates the fact that $w^\prime$ reached the decision step in line 12, a contradiction.

Lemma 3.1.6. All the new group names are from the range $[\ell]$, where $\ell = 2m - 1$.

Proof. In what follows, we prove that the proposal value of any processor in any iteration is in the range $[\ell]$. Clearly, this proves the lemma as the chosen name of any group is a proposal value of some processor. Consider some processor. It is clear that its first iteration proposal value is in the range $[\ell]$. Thus, let us consider some iteration $k > 1$ and prove that its proposal value is at most $2m - 1$. Essentially, this is done by bounding the value of $p$ calculated in line 11 of the preceding iteration. For this purpose, we first claim that the set $P$ consists of at most $m - 1$ values. Notice that $P$ holds the proposal values of processors from at most $m - 1$ groups. Furthermore, observe that for each of those groups, it holds the proposal values of processors having the same maximal iteration counter. This implies, in conjunction with Lemma 3.1.3, that for each of those groups, the proposal values of the corresponding processors are identical. Consequently, $P$ consists of at most $m - 1$ distinct values. Now, one can easily verify that the rank of every group calculated in line 10 is at most $m$. Therefore, the new value of $p$ is no more than $2m - 1$.

Lemma 3.1.7. Any processor either takes finite number of steps or chooses a new group name.

Proof. The proof of this theorem is a natural generalization of the termination proof of the renaming algorithm (see, e.g., [6, Sec. 16.3]).

Assume, by way of contradiction, that some processor takes an infinite number of steps in execution $\alpha$ without choosing a new group name. Considering the proof of Lemma 3.1.4, all the non-faulty
processors from this group will take an infinite number of steps without choosing a new group name; we say that such a group is trying. The processor that wins the consensus object corresponding to its group at the most advanced iteration that was executed by processors from that group, is referred to as the group representative; hence, we may refer to the processor’s operations as the group’s operations and will satisfy in proving that the group takes a finite number of steps or chooses a new group name.

Consider a finite prefix of $\alpha$ such that all trying groups have already updated the snapshot object at least once (by executing Line 4) and all other groups have either chosen a new group name or taken all their steps. Denote by $\alpha'$ the remaining suffix of $\alpha$; note that only trying groups take steps in $\alpha'$. Let $G_i$ be the trying group with the smallest original group name; we argue that $G_i$ decides in $\alpha'$, which is a contradiction. Let $NF$ (for ”not free”) be the set of names appearing in the atomic snapshot object at the beginning of $\alpha'$ who were proposed by the representatives of the ”not trying” groups; note that this set remains fixed in $\alpha'$. Let $F$ (for ”free”) be the complementary set, that is, $F = [1...2m-1] - NF$; assume that $F = \{z_1, z_2, ..\}$, where $z_1 < z_2 < ....$

Consider a point in $\alpha'$ where every trying group has written its proposal value based on a view returned by a snapshot scan that started in $\alpha'$. Since no group performs Line 4 for the first time in $\alpha'$, it follows that all views contain the same set of original group names; therefore, each group gets a distinct rank. Let $r$ be the rank of $G_i$’s original group name in this view, bear in mind that we assumed that $G_i$ has the smallest rank among all the trying groups. We first argue that no trying group other than $G_i$ ever proposes a name in $\{z_1, ..., z_r\}$ once every trying group has updated the snapshot object in Line 4, based on a scan that started in $\alpha'$. Consider another trying group $G_j$. When $G_j$’s representative performs a snapshot scan in $\alpha'$, it sees every name used in $NF$ and possibly some other names as well. Thus the free group names from $G_j$’s perspective form a set $F' \subseteq F$. Since $G_j$’s original group name has rank greater than $r$, $G_j$ proposes a group name greater than $z_r$. We now argue that $G_i$ will eventually propose $z_r$ in $\alpha'$. If not, then $G_i$ always sees $r$ as someone else’s proposal. By definition, $z_r$ is not a member of $NF$. Thus it is continually proposed by other trying groups. But by the previous claim, every other trying group will eventually get to a point in time after which it only proposes greater names. Thus eventually $G_i$ will stop seeing $z_r$ as someone else’s proposal. Thus eventually $G_i$ will propose $z_r$, see no conflicting proposal of $z_r$,
3.2 An impossibility result

We provide an FLP-style proof of the following theorem.

**Theorem 3.2.1.** For any $g \geq 2$, it is impossible to wait-free implement tight group renaming in a system having $(g - 1)$-consensus objects and atomic registers.

In particular, Theorem 3.2.1 implies that there is no wait-free implementation of tight group renaming, even when $g = 2$, using only atomic registers.

In what follows, we establish the proof of Theorem 3.2.1. Our impossibility proof follows the high level FLP-approach employed in the context of the consensus problem (see, e.g., [13, 10]). Namely, we assume the existence of a tight group renaming algorithm, and then derive a contradiction by constructing a sequential execution in which the algorithm fails, either because it is inconsistent, or since it runs forever. Prior to delving into technicalities, we introduce some terminology.

The decision value of a processor is the new group name selected by that processor. Analogously, the decision value of a group is the new group name selected by all processors of that group. An algorithm state is multivalent with respect to group $G$ if the decision value of $G$ is not yet fixed, namely, the current execution can be extended to yield different decision values of $G$. Otherwise, it is univalent. In particular, an $x$-valent state with respect to $G$ is a univalent state with respect to $G$ yielding a decision value of $x$. A decision step with respect to $G$ is an execution step that carries the algorithm from a multivalent state with respect to $G$ to a univalent state with respect to $G$. A processor is active with respect to an algorithm state if its decision value is still not fixed. A algorithm state is critical with respect to $G$ if it is multivalent with respect to $G$ and any step of any active processor is a decision step with respect to $G$.

**Lemma 3.2.2.** Every group renaming algorithm admits an input instance whose initial algorithm state is multivalent with respect to a maximal size group.

**Proof.** We begin by establishing that every group renaming algorithm admits an input instance
whose initial algorithm state is multivalent with respect to some group. Consider some group renaming algorithm, and assume by contradiction that the initial algorithm state is univalent with respect to all groups for every input instance. We argue that all processors implement some function \( f : [M] \rightarrow [\ell] \) for computing their new group name. For this purpose, consider some processor whose group name is \( a \in [M] \). Notice that this processor may be scheduled to execute a “solo run”. Let us assume that its decision value in this case is \( x \in [\ell] \). Since the initial algorithm state is univalent with respect to the group of that processor, it follows that in any execution this processor must decide \( x \), regardless of the other groups, their name, and their scheduling. The above-mentioned argument follows by recalling that all processors execute the same algorithm, and noticing that \( a \) could have been any initial group name. Now, recall that \( M > \ell \). This implies that there are at least two group names \( a_1, a_2 \in [M] \) such that \( f(a_1) = f(a_2) \). Correspondingly, there are input instances in which two processors from two different groups decide on the same new group name, violating the uniqueness property.

We now turn to prove that every group renaming algorithm admits an input instance whose initial algorithm state is multivalent with respect to a maximal size group. Consider some group renaming algorithm, and suppose its initial algorithm state is multivalent with respect to group \( \mathcal{G} \). Namely, there are two execution sequences \( \sigma_1, \sigma_2 \) that lead to different decision values of \( \mathcal{G} \). Now, if \( \mathcal{G} \) is maximal in size then we are done. Otherwise, consider the input instance obtained by adding processors to \( \mathcal{G} \) until it becomes maximal in size. Notice that the execution sequences \( \sigma_1 \) and \( \sigma_2 \) are valid with respect to the new input instance. In addition, observe that each possessor must decide on the same value as in the former instance. This follows by the assumption that none of the processors has prior knowledge about the other processors and groups, and thus each processor cannot distinguish between the two instances. Hence, the initial algorithm state is also multivalent with respect to \( \mathcal{G} \) in this new instance.

\begin{lemma}
Every group renaming algorithm admits an input instance for which a critical state with respect to a maximal size group may be reached.
\end{lemma}

\begin{proof}
We prove that every group renaming algorithm which admits an input instance whose initial algorithm state is multivalent with respect to some group may reach a critical state with respect to
\end{proof}
that group. Notice that having this claim proved, the lemma follows as consequence of Lemma [3.2.2].

Consider some group renaming algorithm, and suppose its initial algorithm state is multivalent with respect to group $G$. Consider the following sequential execution, starting from this state. Initially, some arbitrary processor executes until it reaches a state where its next operation leaves the algorithm in a univalent state with respect to $G$, or until it terminates and decides on a new group name. Note that the latter case can only happen if the underlying processor is not affiliated to $G$. Also note that the processor must eventually reach one of the above-mentioned states since the algorithm is wait-free and cannot run forever. Later on, another arbitrary processor executes until it reaches a similar state, and so on. This sequential execution continues until reaching a state in which any step of any active processor is a decision step with respect to $G$. Again, since the algorithm cannot run forever, it must eventually reach such state, which is, by definition, critical.

We are now ready to prove the impossibility result.

**Proof of Theorem 3.2.1.** Assume that there is a group renaming algorithm implemented from atomic registers and $r$-consensus objects, where $r < g$. We derive a contradiction by constructing an infinite sequential execution that keeps such algorithm in a multivalent state with respect to some maximal size group. By Lemma [3.2.3], we know that there is an input instance and a corresponding execution of the algorithm that leads to a critical state $s$ with respect to some group $G$ of size $g$. Keep in mind that there are at least $g$ active processors in this critical state since, in particular, all the processors of $G$ are active. Let $p$ and $q$ be two active processors in the critical state which respectively carry the algorithm into an $x$-valent and a $y$-valent states with respect to $G$, where $x$ and $y$ are distinct. We now consider four cases, depending on the nature of the decision steps taken by the processors:

**Case I: One of the processors reads a register.** Let us assume without loss of generality that this processor is $p$. Let $s'$ be the algorithm state reached if $p$’s read step is immediately followed by $q$’s step, and let $s''$ be the algorithm state following $q$’s step. Notice that $s'$ and $s''$ differ only in the internal state of $p$. Hence, any processor $p' \in G$, other than $p$, cannot distinguish between these states. Thus, if it executes a “solo run”, it must decide on the same value. However, an
impossibility follows since \( s' \) is \( x \)-valent with respect to \( G \) whereas \( s'' \) is \( y \)-valent. This case is schematically described in Figure 3.1(a).

**Case II: Both processors write to the same register.** Let \( s' \) be the algorithm state reached if \( p \)'s write step is immediately followed by \( q \)'s write step, and let \( s'' \) be the algorithm state following \( q \)'s write step. Observe that in the former scenario \( q \) overwrites the value written by \( p \). Hence, \( s' \) and \( s'' \) differ only in the internal state of \( p \). Therefore, any processor \( p' \in G \), other than \( p \), cannot distinguish between these states. The impossibility follows identically to Case I.

**Case III: Each processor writes to or competes for a distinct register or consensus object.** In what follows, we prove impossibility for the scenario in which both processors write to different registers, noting that impossibility for other scenarios can be easily established using nearly identical arguments. The algorithm state that results if \( p \)'s write step is immediately followed by \( q \)'s write step is identical to the state which results if the write steps occur in the opposite order. This is clearly impossible as one state is \( x \)-valent and the other is \( y \)-valent. This case is schematically illustrated in Figure 3.1(b).

**Case IV: All active processors compete for the same consensus object.** As mentioned above, there are at least \( g \) active processors in the critical state. Additionally, we assumed that the algorithm uses \( r \)-consensus objects, where \( r < g \). This implies that the underlying consensus object is accessed by more processors than its capacity, which is illegal.

![Figure 3.1: The decision steps cases.](image-url)
Chapter 4

Loose Group Renaming

In this section, we restrict our attention to shared memory systems which support only atomic registers. By Theorem 3.2.1, we know that it is impossible to solve the tight group renaming problem unless we relax our goal. Accordingly, we consider a variant in which processors from the same group may choose different new group names, as long as the uniqueness property is maintained (i.e., processors from different groups have distinct names). The objective is to minimize the range of the new group names, denoted by $\ell$, while maintaining the number of new group names selected by processors from the same group minimal. We use the notation $\ell$-group renaming algorithm to designate such an algorithm for new names selection out of the range $\{1, \ldots, \ell\}$.

4.1 The non-uniform case

In the following, we consider the loose group renaming task without any assumptions on the group sizes. We try to settle, to some extent, an open question posed by Gafni [11], which called for an adaptive group renaming algorithm that requires at most $m(m + 1)/2$ names on one extreme, and no more than $2n - 1$ names on the other. Gafni has posed the problem for input instances satisfying the condition $n = m \cdot g$, or in other words for groups of uniform size. We expend the scope of this problem and investigate this task under broader conditions.

We present an adaptive $\ell$-group renaming algorithm having a name selection from the range
where \( \ell = 4n - 3 \) on the worst case scenario. The algorithm also guarantees that no more than \( \min\{g, 2m, 3\sqrt{n} - 1\} \) new group names are chosen by processors within the same group.

The algorithm execution is adjusting with respect to an estimation of the input properties (i.e., the number of processors \( n \), the number of groups \( m \), and the maximal group size \( g \)) to yield better performance results for each instance of the task. A good solution is supplied both for input instances with a small number of large groups and for input instances with a large number of small groups, and for any possible combination in between. Namely, the algorithm guarantees good performance for every input instance, based on an estimation of the number of groups, the number of processors and the maximal group size. The algorithm is built upon a consolidation of two algorithms, denoted Algorithm 2 and Algorithm 3. Both algorithms are adaptations of previously known renaming methods for groups (see, e.g., [11]) and are fully described in [1].

Algorithm 2 code for processor \( i \in [N] \).

In shared memory: \( SS[1, \ldots, N] \) array of swmr registers, initially \( \perp \).

1: \( p_i \leftarrow 1 \)
2: \textbf{while } true
3: \( SS[i] \leftarrow \langle \text{GID}_i, p_i \rangle \)
4: \( (\langle \text{GID}_1, p_1 \rangle, \ldots, \langle \text{GID}_N, p_N \rangle) \leftarrow \text{Snapshot}(SS) \)
5: \textbf{if } \( p_i = p_j \) for some \( j \in [N] \) having \( \text{GID}_j \neq \text{GID}_i \)
6: \( r \leftarrow \text{the rank of } \text{GID}_i \text{ in } \{ \text{GID}_k \neq \perp : k \in [N] \} \)
7: \( p_i \leftarrow \text{the } r\text{-th integer not in } \{ p_k \neq \perp : k \in [N] \setminus \{i\} \} \)
8: \textbf{else } return \( p_i \)
9: \textbf{end if}
10: \textbf{end while}

Algorithm 3 code for processor \( i \in [N] \).

In shared memory: \( SS[1, \ldots, N] \) array of swmr registers, initially \( \perp \).

1: \( SS[i] \leftarrow \text{GID}_i \)
2: \( (\text{GID}_1, \ldots, \text{GID}_N) \leftarrow \text{Snapshot}(SS) \)
3: \( \tilde{m} \leftarrow \text{the number of distinct GIDs in } \{ \text{GID}_j \neq \perp : j \in [N] \} \)
4: \( r \leftarrow \text{the rank of } \text{GID}_i \text{ in } \{ \text{GID}_j \neq \perp : j \in [N] \} \)
5: return \( \tilde{m}(\tilde{m} - 1)/2 + r \)

Algorithm 2 is a \( \{n + m - 1\} \)-group renaming algorithm that efficiently handles small-sized groups. It guarantees the selection of up to \( g \) new group names by processors within the same group. Every processor that executes Algorithm 2 keeps efficiently proposing a new group name in the range \( \{1, \ldots, n + m - 1\} \) that has not been proposed by processors from other groups yet, based on a
local snapshot. Following the new group name proposal, the processor takes a new snapshot view and decided, based on the updated snapshot, whether to accept the proposed group name and return or whether to continue proposing a new group name. Algorithm 3 is a \( m(m+1)/2 \)-group renaming algorithm that efficiently handles a small number of groups. It guarantees the selection of up to \( \min\{m,g\} \) new group names by processors within the same group. Every processor that executes Algorithm 3 chooses, based on its snapshot view, a new group name from a range of names unique to its group, such that every group has a set of unique names.

We are now ready to present our \textit{n-independent} adaptive loose group renaming algorithm. The algorithm has its roots in the natural approach that applies the best response with respect to the input instance. For example, Algorithm 3 allows the choice of up to \( \min\{m,g\} \) different names within the same group, while Algorithm 2 allows the choice of up to \( g \) different names within the same group. It is easy to see that Algorithm 3 outperforms Algorithm 2 with respect to the number of new group names selected within the same group, as \( \min\{m,g\} \leq g \), for any instance. In addition, one can verify that when \( m < \sqrt{n} \), Algorithm 3 selects new group names from the range \( \{1, \ldots, \ell\} \) with \( \ell \leq (n/2 - \sqrt{n}/2) \), whereas Algorithm 2 selects name from the range \( \{1, \ldots, \ell\} \) with \( (\ell \geq n) \). Hence, given an instance having \( m < \sqrt{n} \), the best attitude would be to execute Algorithm 3. Unfortunately, a straight-forward application of this approach has several difficulties. One immediate difficulty concerns the implementation since none of the processors have prior knowledge of the real values of \( n \), \( m \) or \( g \). Our algorithm bypasses this difficulty by estimating these parameters (i.e., estimated number of processors - \( \tilde{n} \), estimated number of groups - \( \tilde{m} \) and an estimated group size - \( \tilde{g} \)) using an atomic snapshot object. Another difficulty concerns with performance issues, since both algorithms show poor guarantees regarding the number of new group names selected within the same group for instances which simultaneously satisfy \( g = \Omega(n) \) and \( m = \Omega(n) \). One concrete example having \( g = n/2 \) and \( m = n/2 + 1 \) consists of a single group having \( n/2 \) members and \( n/2 \) singleton groups. In this case, up to \( n/2 \) different names may be chosen by processors within the same group both on Algorithm 2 and Algorithm 3 execution. We overcome this difficulty by using an educated combination of the algorithms, yielding a name selection guarantee of \( 3\sqrt{n} - 1 \) names within the group for these “hard” cases. The key observation utilized in this context is that if there are many groups then most of them must be small (i.e. of size not greater than \( \sqrt{n} \)). Consequently, by filtering out the small-sized groups (using Algorithm 2
in most cases - where the estimated number of groups is $\sqrt{n}$ or greater), we are left with a small number of large groups that we can handle efficiently.

**Algorithm 4** Adjusting group renaming algorithm: code for processor $i \in [N]$.

In shared memory: $SS[1, \ldots, N]$ array of swmr registers, initially $\bot$.

1: $SS[i] \leftarrow GID_i$
2: $(GID_1, \ldots, GID_N) \leftarrow \text{Snapshot}(SS)$
3: $\tilde{n} \leftarrow$ the total number of processors in $SS$
4: $\tilde{m} \leftarrow$ the number of distinct GIDs in $\{GID_j \neq \bot : j \in [N]\}$
5: $\tilde{g} \leftarrow$ the number of processors $j \in [N]$ having $GID_j = GID_i$
6: if $(\tilde{m} < \sqrt{\tilde{n}}) \parallel (\tilde{g} > \sqrt{\tilde{n}})$
7: /* Processor $i$ executes Algorithm 3 if there is a small */
8: /* number of groups or it is a member of a large group */
9: $x \leftarrow$ the outcome of Algorithm 3 (using shared memory $SS_1[1, \ldots, N]$)
10: return $2x$
11: else /* Processor $i$ executes if there is a large number of small groups */
12: $x \leftarrow$ the outcome of Algorithm 2 (using shared memory $SS_2[1, \ldots, N]$)
13: return $2x - 1$
14: end if

**Theorem 4.1.1.** Algorithm 4 is a $\{4n - 3\}$-group renaming algorithm with a guarantee of up to $\min\{g, 2m, 3\sqrt{n} - 1\}$ group names selection within the same group.

Proof. We begin by establishing the correctness of the algorithm, by showing it maintains the uniqueness property and terminates after a finite number of steps. One can easily validate that the termination property holds since both Algorithm 2 and Algorithm 3 terminate after finite number of steps. As for the uniqueness property, recall that both Algorithm 2 and Algorithm 3 maintain the uniqueness property, and notice that each case of the if statement (see lines 6-15) utilizes a distinct set of new names. In the first case of the if statement any processor that executes Algorithm 3 in line 9, is assigned with a new name in the range $\{\text{evens}|1, \ldots, 4n - 2\sqrt{n}\}$. This observation follows from the name space properties of Algorithm 3 (satisfying $\ell \leq \{m(m + 1)/2\}$) combined with the statement in claim 4.1.4, which states that processors from at most $2\lfloor\sqrt{n}\rfloor - 1$ groups may execute Algorithm 3. Continuing with the second case of the if statement, any processor that executes Algorithm 2 in line 13 is assigned with a new name in the range $\{\text{odds}|1, \ldots, 4n - 3\}$. This observation follows from the name space properties of Algorithm 2 (satisfying $\ell \leq \{n + m - 1\}$), combined with the observation that $m \leq n$.  

23
We now turn to establish the performance properties of the algorithm. We demonstrate that it is adaptive (or in other words applies the best response with respect to changing input conditions) both to $\ell$, the total number of new group names selected by all groups, and both to the number of new group names that may be selected by processors within the same group.

$\ell$ maintains the following properties:

\[
\begin{cases}
\{1, \ldots, 2m^2 + 2m - 3\} & m < \sqrt{n} \\
\{1, \ldots, 4n - 3\} & m \geq \sqrt{n} \text{ and } g \leq \sqrt{n} \\
\{1, \ldots, 4n - 3\} & m \geq \sqrt{n} \text{ and } g > \sqrt{n}
\end{cases}
\]

The number of new group names selected by each group follows the restrictions:

\[
\begin{cases}
\min\{g, 2m\} & m < \sqrt{n} \\
g & m \geq \sqrt{n} \text{ and } g \leq \sqrt{n} \\
\min\{g, 3\sqrt{n} - 1\} & m \geq \sqrt{n} \text{ and } g > \sqrt{n}
\end{cases}
\]

**Case I:** $m < \sqrt{n}$. The estimation values never exceed their real values, namely, $\tilde{n} \leq n$, $\tilde{m} \leq m$ and $\tilde{g} \leq g$. Some processors may execute Algorithm 3 in line 9 and some may execute Algorithm 2 in line 13 depending on the concrete execution sequence. The execution of Algorithm 3 guarantees the choice of at most $\min\{m, g\}$ names for each group. These new names are even numbers from the range $\{1, \ldots, m(m+1)\}$. The execution of Algorithm 2 guarantees the selection of up to $g \leq m$ names within each group. These new names are odd numbers from the range $\{1, \ldots, 2m^2 + 2m - 3\}$.

Summing it all together, Algorithm 4 guarantees the selection of up to $\min\{g, 2m\}$ names for each group from the range $\{1, \ldots, 2m^2 + 2m - 3\}$. Notice that $\min\{g, 2m\} \leq \min\{g, 2m, 3\sqrt{n} - 1\}$ and $(2m^2 + 2m - 3) \leq (4n - 3)$ since $(m < \sqrt{n})$. Thus, the performance properties of the algorithm in this case support the worst case analysis.

**Case II:** $m \geq \sqrt{n}$ and $g \leq \sqrt{n}$. Every processor may execute any of the algorithms, depending
on the concrete execution sequence. The number of new group names selected by processors within
the same group is trivially \( g \) since \( g \) is the maximal group size. Furthermore, it can be shown that
the size of the name space \( \ell \) does not exceed \( 4n - 3 \). It is done by a simple interleaving of the
name space that may be used by Algorithm 3 which is \{evens\}|\{1, \ldots, 4n - 2\}\sqrt{n}\} since at most
\( 2\sqrt{n} - 1 \) groups may execute it, following claim 4.1.4, with the name space that may be used by
Algorithm 2 which is \{odds\}|\{1, \ldots, 4n - 3\}. Notice that \( g \leq \min\{g, 2m, 3\sqrt{n} - 1\} \) since \( g \leq \sqrt{n} \leq m \),
and that \( 4n - 3 \) is the maximal size of the name space of Algorithm 4. Hence, the performance
properties of the algorithm in this case support the worst case analysis.

**Case III: m \geq \sqrt{n} and g > \sqrt{n}**. Every processor may execute any of the algorithms, depend-
ing on the concrete execution sequence. The first observation one should make is that no more
than \( 2\sqrt{n} - 1 \) new names from the range \{evens\}|\{1, \ldots, 4n - 2\}\sqrt{n}\} may be collectively assigned to
processors from the same group by Algorithm 3 (since at most \( 2\sqrt{n} - 1 \) groups may execute it, consid-
ering claim 4.1.4). Observing Algorithm 4 definition, only groups of size not greater than
\( \sqrt{n} \) meet the condition for the execution of Algorithm 2, hence allowing processors from the same
group to choose a set of at most \( g \leq \sqrt{n} \) different names from the range \{odds\}|\{1, \ldots, 4n - 3\}. Putting everything together, we attain that each group of processors could be assigned with no
more than \( \min\{g, 3\sqrt{n} - 1\} \) new group names from a range \{1, \ldots, 4n - 3\}. It is easy to see that
\( \min\{g, 3\sqrt{n} - 1\} \leq \min\{g, 2m, 3\sqrt{n} - 1\} \) since \( m \geq \sqrt{n} \), and thus the performance properties of
the algorithm in this case also support the worst case analysis.

In order to complete the proof of theorem 4.1.1 it is required to establish claim 4.1.4 which states
that no more than \( 2\sqrt{n} - 1 \) groups may execute algorithm 3.

**Proposition 4.1.2.** Assume that \( p_i \) is a member in a group that consists of up to \( k + 1 \) processors.
If \( p_i \)'s snapshot view consists of \( \tilde{n} \) processors, \( \tilde{n} \geq (k + 1)^2 \), and more than \( k \) groups, \( p_i \) will not
execute algorithm 3.

**Proof.** \( p_i \) is a member in a group that consists of up to \( k + 1 \) processors and has in its snapshot
view \( \tilde{n} \) processors such that \( \sqrt{\tilde{n}} \geq (k + 1) \). Therefore \( p_i \) fails to satisfy the condition \( (\tilde{g} > \sqrt{\tilde{n}}) \) for
algorithm 3 execution. If there are more than \( k \) groups in the snapshot of this processor, it will also
fail to satisfy the other condition \( \tilde{m} < \sqrt{\tilde{n}} \) for algorithm 3 execution. Thus none of the conditions for algorithm 3 execution will be satisfied.

**Claim 4.1.3.** Consider \( R_{OPT} \), a run that consists of \( \tilde{n} < (h + 1)^2 \) processors and has a maximum number of groups (denoted by \( OPT \)) executing algorithm 3. In \( R_{OPT} \) the total number of processors in groups that do not run algorithm 3 is at most \( h \).

**Proof.** Assume by contradiction that there is such a run, \( R_{OPT}^* \), that consist of \( \tilde{n} < (h + 1)^2 \) processors, has \( OPT \) groups executing algorithm 3 and \( h + 1 \) processors that do not participate in any of these groups. Build a new run, \( R_{OPT}^{\tilde{R}} \), based on \( R_{OPT}^* \) in which the above mentioned \( h + 1 \) processors run in the same group. In \( R_{OPT}^{\tilde{R}} \), the new run that was just built, there is an additional group of size \( h + 1 \). Therefore at least one of the processors in this group satisfies the condition \( (\tilde{g} > \sqrt{\tilde{n}}) \) for algorithm 3 execution. On top of that, each of the groups that have executed algorithm 3 in the original run, \( R_{OPT}^* \), still have one processor or more satisfying at least one of the conditions \( ((\tilde{g} > \sqrt{\tilde{n}}) \text{ or } (\tilde{m} < \sqrt{\tilde{n}})) \) for algorithm 3 execution, due to one of three following reasons:

1. groups that were executing algorithm 3 in \( R_{OPT}^* \) remained unchanged.
2. \( R_{OPT}^{\tilde{R}} \) and \( R_{OPT}^* \) have the same execution order.
3. the number of executing groups did not increase.

If at some stage of \( R_{OPT}^{\tilde{R}} \) execution, in contrast to the parallel scenario in \( R_{OPT}^* \), the number of executing groups has decreased under \( h \), some additional processors might have satisfied the condition \( (\tilde{m} < \sqrt{\tilde{n}}) \) and executed algorithm 3. The rest of the processors, that their snapshot views had more than \( h \) groups followed their behavior in \( R_{OPT}^* \). Therefore we may conclude that each of the groups in the new run, \( R_{OPT}^{\tilde{R}} \), have the same processors executing algorithm 3 as in the \( R_{OPT}^* \) with the possible addition of some new processors that did not execute algorithm 3 before. Summing it all up, the new run, \( R_{OPT}^{\tilde{R}} \), has \( OPT + 1 \) groups executing algorithm 3, in contrast to the maximality assumption.

**Inductive Claim 4.1.4.** For any number of processors \( n \) that execute algorithm 4, no more than \( 2\lfloor \sqrt{n} \rfloor - 1 \) groups execute algorithm 3.
Proof. By induction on \( k \), an integer value in \( \{1, \ldots, \lfloor \sqrt{n} \rfloor \} \).

\( k \) denotes the lower value of the square root of the estimated number of executing processors at any given time during the induction, i.e. \( k = \lfloor \sqrt{n} \rfloor \).

For any integer value \( k \), we will prove that at most \( 2k - 1 \) groups execute algorithm 3.

Step \( k \) begins as the first snapshot that includes at least \( k^2 \) processors was taken. Any processor that executes during this step of the induction has in its snapshot view an estimated number of processors, \( \tilde{n} \), satisfying \( k^2 \leq \tilde{n} \leq (k + 1)^2 - 1 \).

**Base case:** Lets prove the inductive claim for \( k = 1 \).

In the case that \( k = 1 \) less than four processors participate in the algorithm. We show that if less than four processors participate in the execution, at most one group executes algorithm 3.

In the case that there is only one group the base statement is trivially satisfied. If two groups participate then at most one of them has two processors in it and satisfies one of the conditions in line 6 and thus executes algorithm 3. Finally, if three groups participate, none of them satisfy the second condition in line 6. Therefore no more than one group executes algorithm 3, as required.

**Induction hypothesis:** Assume that the inductive claim holds for every natural \( k \leq K \).

Denote by \( \tilde{n}_b \) the estimated number of executing processors. By the inductive hypothesis, if \( \tilde{n}_b < (k + 1)^2 \), then no more than \( 2(K - i) - 1 \) groups execute algorithm 3 where \( i \) is a natural value within the range \( \{0, \ldots, K - 1\} \).

**Inductive step:** Prove for \( k \leq (K + 1) \) that the total number of groups that execute algorithm 3 is at most \( 2(K + 1) - 1 \).

In order to establish our claim we analyze the maximum number of groups that may satisfy each of the conditions in line 6. We divide all possible scenarios into two cases and analyze them separately. *In both cases we refer to processors that have start executing during the inductive step.* We say that a processor executes during the inductive step if it sees in its snapshot view at least \( (K + 1)^2 \) processors.

27
The first scenario we analyze refers to the cases in which at least one processor, that has began its execution during the inductive step satisfies the first condition \( \tilde{m} < \sqrt{\tilde{n}} \) in line 6. The second scenario refers to the rest of the cases in which non of these processors, who have started executing at the inductive step, satisfy this condition.

Note that the induction on \( k \) is not necessarily on consecutive values of \( k \). The inductive step is on \( k = K+1 \) while the inductive assumption is on any natural value \( k = K-i \) where \( i \in \{0, \ldots, K-1\} \).

The algorithm may jump directly from \( k = K-i \) to \( k = K+1 \) since a variable number processors may wake up simultaneously.

It is assumed that the last processor that started executing before the inductive step has less than \((K - i + 1)^2\) processors in its snapshot view. In particular any processor that started executing before the inductive step has in its snapshot view at most \((K + 1)^2 - 1\) processors. Therefore, any processor that started executing during the inductive step has in its snapshot view at least \((K + 1)^2\) processors.

**Case I: There is a processor satisfying the condition \((\tilde{m} < \sqrt{\tilde{n}})\) during the inductive step.**

Consider the maximum number of groups that can run algorithm 3 in an execution that has a processor which has started its execution during the inductive step and satisfies the condition \((\tilde{m} < \sqrt{\tilde{n}})\). Note that by definition the above mentioned processor satisfies the requirement \((\tilde{m} < (K + 1))\). Denote by \( p_j \) the processor that satisfies the requirement \((\tilde{m} < (K + 1))\) and has the largest number of processors in its snapshot view among the processors that satisfy this condition.

We divide the analysis into two parts, before and after the execution of \( p_j \). First, we investigate the maximum number of groups that may start executing algorithm 3 up to the point in time in which \( p_j \) has started its execution. Observe that in \( p_j \)’s snapshot view there are at least \((K + 1)^2\) processors and at most \( K \) groups. Hence, at most \( K \) groups may execute algorithm 3 up to this point. Now we turn to investigate the maximum number of groups that may execute
algorithm 3 after \(p_j\) has taken its snapshot view. Consider that at most \((K + 2)^2 - 1\) processors may execute up to the end of the inductive step (i.e., the time of execution of the last processor that has in its snapshot view \(\tilde{n}_b\) processors satisfying \(\lfloor \sqrt{\tilde{n}_b} \rfloor = K + 1\)). Therefore there are at most \((K + 2)^2 - (K + 1)^2 - 1 = 2K + 2\) additional processors that may have in their snapshots \(\tilde{n}_b\) processors (\(\lfloor \sqrt{\tilde{n}_b} \rfloor = K + 1\)) and did not satisfy the above condition \((\tilde{m} < (K + 1))\). Since any processor executing after \(p_j\) may only satisfy the condition \((\tilde{g} > K + 1)\) in line 6 any group that executes algorithm 3 and is not mentioned above has at least \(K + 2\) processors. Since \(\frac{2K+2}{K+2} < 2\), there is at most one additional group executing algorithm 3. To sum it all up, there are at most \(K + 1\) groups that may execute algorithm 3, which is less than \(2(K+1) - 1\), as required.

Case II: No processor satisfies the condition \((\tilde{m} < \sqrt{\tilde{n}})\) during the inductive step.

Assume that there are \(\tilde{n}\) executing processors such that \(\lfloor \sqrt{\tilde{n}} \rfloor = K + 1\) and also assume that non of the processors satisfy the requirement \((\tilde{m} < K + 1)\) during the inductive step. In other words, any processor (denoted by \(p_i\)) that has started executing during the inductive step, does not satisfy the requirement \((\tilde{m} < K + 1)\). If a processor \(p_i\) executes algorithm 3 during the inductive step it satisfies the condition \(\tilde{g} > k + 1\).

According to the inductive assumption, at the end of that stage there could be at most \((K-i+1)^2-1\) executing processors divided to at most \(2(K-i)-1\) groups that may run algorithm 3. Considering proposition 4.1.2 after the end of the induction assumption stage, the number of groups of size less than or equal to \((K + 1)\) that execute algorithm 3 would not increase. Therefore it is enough to bound the maximum number of groups of size equal or larger than \((K + 2)\) that have started executing algorithm 3 during the inductive step. In other words, it is enough to bound the number of groups of size equal or greater than \((K + 2)\) that do not have any processor executing algorithm 3 during the induction hypothesis stage, but have at least one processor executing algorithm 3 during the inductive step. We call these groups, ”blowing groups”. In order to bound the number of ”blowing groups” it is sufficient to bound the total number of processors available to participate in these groups during algorithm 4 execution. We first bound the maximum number of the available processors which were executing before the inductive step, leaning on claim 4.1.3 Then we turn to bound the maximum number of available processors which were executing during the inductive
In order to prove the bound for the inductive hypothesis step, let $R_{OPT}$ be an optimal execution that maximizes the number of groups executing algorithm 3 during the inductive hypothesis stage. According to claim 4.1.3, in $R_{OPT}$ there are at most $K - i$ processors which participate in groups that do not have any processor executing algorithm 3 during the induction hypothesis stage. In order to prove the bound for the inductive step, consider that there are at most $(K + 2)^2 - 1$ executing processors at that stage, since the induction is on $k = K + 1$ (where $k = \sqrt{\tilde{n}}$). Also consider that in the inductive hypothesis stage at most $(K - i + 1)^2 - 1$ processors have executed. Therefore the contribution of the inductive step to the “blowing groups” is bounded by $(K + 2)^2 - (K - i + 1)^2 - 1$.

To sum it up, at most $(K + 2)^2 - (K - i + 1)^2 + (K - i) - 1$ processors can participate in the “blowing groups”. Since every group that have started executing algorithm 3 during the inductive stage is of size $K + 2$ or greater, there are at most $(K + 2)^2 - (K - i + 1)^2 + (K - i) - 1$ such groups. Considering the inductive hypothesis, the maximal number of groups that could have run algorithm 3 during the execution of algorithm 4 is $2(K - i) + \frac{(K + 2)^2 - [(K - i) + 1]^2 + (K - i) - 1}{K + 2} - 1$. This expression equals to $2(K - i) + \frac{(K^2 + 4K + 4 - [(K - i)^2 + 2(K - i) + 1] + (K - i) - 1}{K + 2} - 1 = 2(K - i) + \frac{(2K + 4)(2iK + 4 - 2i^2 - 1 + K - i - 1}{K + 2} - 1 = 2(K - i) + \frac{2(K + 2) + 2iK + 4 - 2i^2 - 1}{K + 2} - 1 = 2(K + 1) + \frac{(K - i^2 - 2i - 2)}{K + 2} - 1$. Since $(K - i^2 - 2i - 2) < 1$, the maximal number of groups that could have run algorithm 3 is not more than $2(K + 1) - 1$, as required.

### 4.2 The uniform case

In what follows, we study the problem when the groups are guaranteed to be uniform in size. We first refine the worst case analysis of Algorithm 4 under the uniformity assumption. Under the new conditions, $\ell \leq 4n - 3$ and no more than $(\min\{g, 2m\} \leq 2\sqrt{n})$ new group names are chosen by processors within the same group. In particular, we demonstrate that the algorithm is self-adjusting as the following restrictions are maintained:
1. The total number of new group names $\ell$: 

$$\begin{align*}
\ell &= \begin{cases} 
2m(m+1) - 3 & m < \sqrt{n} \\
4n - 3 & m \geq \sqrt{n}
\end{cases}
\end{align*}$$

2. The number of new group names chosen by processors within the same group:

$$\begin{align*}
\ell &= \begin{cases} 
\min\{g, 2m\} & m < \sqrt{n} \\
g & m \geq \sqrt{n}
\end{cases}
\end{align*}$$

This result settles, to some extent, an open question posed by Gafni [11], which called for a self-adjusting group renaming algorithm that requires a total of at most $m(m+1)/2$ names on one extreme, and no more than $2n - 1$ names on the other.

The key observation required to establish this refinement is that $n = m \cdot g$ when the groups are uniform in size. Consequently, either $m < \sqrt{n}$ or $g \leq \sqrt{n}$. Now, the proof of the performance properties follows the same line of argumentation presented in the proof of Theorem 4.1.1.
Chapter 5

Discussion

This work has considered and investigated the tight and loose variants of the group renaming problem. Below we discuss few ways in which our results can be extended. An immediate open question is whether a $g$-consensus task can be constructed from group renaming tasks for groups of size $g$, in a system with $g$ processes. Another question is to design an adaptive group renaming algorithm in which a processor is assigned a new group name, from the range 1 through $k$ where $k$ is a constant multiple of the contention (i.e., the number of different active groups) that the processor experiences. We have considered only one-shot tasks (i.e., solutions that can be used only once), it would be interesting to design long-lived group renaming algorithms. We have focused in this work mainly on reducing the new name space as much as possible, it would be interesting to construct algorithms also with low space and time (step) complexities. Finally, the $k$-set consensus task, a generalization of the consensus task, enables for each processor that starts with an input value from some domain, to choose the input of some participating processor as its output, such that all processors together may choose no more than $k$ distinct output values. It is interesting to find out what type of group renaming task, if any, can be implemented using $k$-set consensus tasks and registers.
Bibliography


