DIMACS Technical Report 94-33
November 1994

Optimal Space Distributed Order-Preserving Lists[1]

by

Michael Saks
Department of Mathematics
Rutgers University
New Brunswick, NJ 08903

Fotios Zaharoglou
Department of Computer Science and Engineering
University of California San Diego
La Jolla, California 92093

DIMACS is a partnership of Rutgers University, Princeton University, AT&T Research, Bellcore, and Bell Laboratories.
DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

Abstract

A move-to-front list and the more general move-to-p list are distributed data objects that provide an abstraction of a temporal ordering on a set of processes in a distributed system. We present a lower bound and a matching upper bound of $\Theta(log^2 n)$ bits on the space per processor needed to implement each of them using single-writer multiple-reader registers.

Keywords: Data structures, distributed systems.

[1]This work was supported in part by NSF under grants CCR89-11388 and CCR92-15293, and by UCSD under Academic Senate grant RQ 236-G. A preliminary version of this paper appeared in the Proceedings of the 10th ACM Symposium on Principles of Distributed Computing, August 1991.

1 Introduction

Implementing data structures in a distributed system typically involves many complications that are not encountered in a centralized system. One such difficulty arises in situations where, because of hardware design or network configuration, each process only has access to a restricted set of registers. One well studied situation where this restriction occurs is shared memory systems consisting of *single writer* registers. In a data structure consisting of such registers, information in a particular register can become old, if the writer to that register is idle for a period of time. The task is to organize the data structure in such a way that it can extract the current information from the register values.

In this paper we investigate a simple distributed data object called a *move-to-front* list. A move-to-front list maintains a permutation of objects $1, 2, \ldots, n$ with two operations: *report* which returns the permutation and *move-to-front(i)* which changes the permutation by moving object i to the beginning of the permutation. In our case the objects correspond to the processors of the system and the operation *move-to-front(i)* can only be performed by processor i.

The move-to-front list is of interest in distributed computation because it can maintain a precedence order of the active jobs in the distributed system. This is a basic tool for breaking symmetry among jobs in distributed systems and is used in many distributed protocols.

Israeli and Li [IL87] proposed Time-Stamp Systems (TSS) as a means of maintaining such an order. Their construction provides an implementation of move-to-front list in which each processor has a *single writer multiple reader (swmr)* register of $O(n)$ bits. Furthermore, they show that, in their framework of TSS, any implementation requires at least this much space. However their TSS is a stronger primitive than move-to-front list and it is not apparent that their lower bound holds for arbitrary implementations of a move-to-front list.

As we will show, move-to-front lists admit distributed implementations that use considerably less space than time-stamp systems. More precisely we provide matching upper and lower bounds of $\Theta(\log^2 n)$ on the space per processor needed to implement move-to-front lists. Independently Azar et al [AK$^+$91] have obtained the same upper bound.

Our implementation is based on the representation of a list by its successor relation. This consists of the set of pairs (a, b) where a precedes b in the list. A list of length n is specified by $n-1$ such pairs. Each such pair needs $O(\log n)$ bits, so in principle one might hope to be able to represent the permutation this way if each processor has an $O(\log n)$ bit register. Because of the single writer restriction on a register access, the pairs stored in the register of a processor that is idle for a long period may become invalid, and it will be necessary to distinguish the invalid pairs from the valid ones. We show that it is possible to do this if we allow each processor to store $3 \log n$ such pairs.

Our implementation relies on the assumption that move-to-front operations do not overlap in time with other operations. Removing this assumption seems to greatly increase the difficulty of the problem; in fact even formulating the problem precisely is a nontrivial undertaking. For time-stamp systems, Dolev and Shavit [DS89] were the first to provide both a precise formulation and an implementation using $O(n)$ bit registers. Their construction though is very complicated. Dwork and Waarts [DW92] gave recently a simple and efficient construction that uses $O(n \log n)$ bits per processor register.

Section 2 contains a precise statement of the problem and our assumptions. The proof of the lower bound is in section 3. An efficient implementation is described in section 4. In section 5 we introduce a generalization of move-to lists and provide an efficient implementation. Finally in section 6 we discuss the results and related open problems.

2 The Problem

A move-to-front list is a shared data object that maintains a permutation of a set, say $\{1, 2, \ldots, n\}$, and supports two operations.

- *report* : returns the permutation
- *move-to-front(i)* : changes the permutation by moving element i in front of the list.

Since the elements in the list are in a one-to-one correspondence with the processors, it is convenient to view the list as a list of processors.

We consider distributed implementations of a list under the following assumptions.

1. Any processor can execute a *report* operation.
2. Only processor i can execute *move-to-front(i)* operation.
3. The implementation uses n *swmr* registers, one for each processor. Processor i is referred to as the *owner* of register i, and is the only processor that can write to that register. Any processor can read any register. (The assumption that there is only one register per processor is inessential and is just made for ease of discussion.)
4. No other operation is executed during the execution of any *move-to-front(i)* operation.

There is an easy implementation of the above data object, in which each processor i has a $n-1$ bit swmr register r_i, whose bits are indexed by the other processors. Consider the processors as nodes in a dynamic directed acyclic complete graph, where at any time there is an edge from i to j if i precedes j in the permutation. The direction of an edge (i, j) where $i < j$ is represented as follows: if $r_i[j] \oplus r_j[i] = 0$ then $i \rightarrow j$ else $i \leftarrow j$. When performing move-to-front(i), processor i sets its bits so that all edges point away from it.

If we modify this scheme slightly by increasing the register size to $n - 1 + \log n$ bits so that each processor can permanently store its id, we have a scheme in which the relative order of any two processors can be determined using only their two registers. Israeli and Li [IL87] proved that any implementation satisfying this stronger requirement needs n bit registers, so the above scheme is nearly optimal under this assumption. As we will see, the improvements to $O(\log^2 n)$ that we get here, come from eliminating this requirement. First, however, we will demonstrate that $\Omega(\log^2 n)$ register size is best possible.

3 The Lower bound

By simple information theoretic considerations the sum of the register sizes in any implementation of a move-to-front list must be at least $\Omega(\log n!) = \Omega(n \log n)$, since we must be able to represent all $n!$ permutations. Here we show that this can be improved.

Theorem 3.1 (with R. Cori) *Consider an arbitrary implementation of move-to-front list under the assumptions of the previous section. Let $r_1, r_2, \ldots, r_n$ be, respectively, the sizes of the registers owned by processors $1, 2, \ldots, n$. Then $r_1+r_2+\cdots+r_n$ is in $\Omega(n \log^2 n)$, i.e., the average size of the n registers is at least $\Omega(\log^2 n)$.*

Proof. Consider an implementation of a move-to-front list. The content of the register of processor i at a given time will be referred to as a *label* of that processor. Let $\Lambda_{i,n}$ be the set of all possible labels (register values), that processor i can have. Let $\lambda_{i,n} = |\Lambda_{i,n}|$. Clearly $\lambda_{i,n} \leq 2^{r_i}$. A *global memory state* is a tuple $l^{(n)} = (l_1, l_2, \ldots, l_n)$ where $l_i \in \Lambda_{i,n}$. Let Λ_n be the set of all possible global memory states. Clearly $|\Lambda_n| \leq \prod_{i=1}^{n} \lambda_{i,n}$.

Define $L(k)$ for each integer k to be the minimum number of memory states for any implementation of move-to-front list for k processors, where we assume $L(1) = 1$. Our goal is to prove a lower bound on $L(n)$. Consider the implementation for which $|\Lambda_n| = L(n)$.

Each of the memory states in Λ_n specifies a unique permutation. Let $T_n^{(i)} = \{\sigma \mid \sigma_n = i\}$ be the set of all permutations whose last element is i. The sets $T_n^{(i)}$ partition S_n, the set of all permutations of n objects. Let $\Lambda_n^{(i)}$ be the set of all memory states corresponding to permutations in $T_n^{(i)}$. Then $L(n) \geq \sum_i |\Lambda_n^{(i)}|$. We now proceed to lower bound $|\Lambda_n^{(i)}|$.

Fix a processor i, and let $v \in \Lambda_{i,n}$ be an arbitrary value that can appear in register i. Choose an operation sequence which leads to a memory state in which v is in register i. Beginning from this state, have each processor other than i perform a move-to-front operation. This leads to a permutation in $T_i(n)$, while the value in register i is still v. Now, starting from this state, consider all possible sequences of operations in which processor i never executes another operation. Then all future permutations will belong to $T_i^{(n)}$ and, in addition, the value written in register i remains at v. Since all permutations in $T_i^{(n)}$ can occur, the number of distinct memory states for the system consisting of the remaining $n-1$ processors must be at least $L(n-1)$.

Since v could be any of $\lambda_{i,n}$ possible values, we conclude that the set $\Lambda_n^{(i)}$ must have at least $\lambda_{i,n} L(n-1)$ distinct memory states. Summing over i, we get:

$$L(n) = \sum_{i=1}^{n} |\Lambda_n^{(i)}| \geq \sum_{i=1}^{n} \lambda_{i,n} \; L(n-1)$$

Let $\lambda_n = (1/n) \sum_{i=1}^{n} \lambda_{i,n}$. Since $L(n) \leq \prod_{i=1}^{n} \lambda_{i,n}$, the arithmetic - geometric mean inequality implies $\lambda_n \geq L(n)^{1/n}$. The recurrence relation becomes

$$L(n) \geq n \; L(n)^{1/n} \; L(n-1)$$

or

$$L(n) \geq n^{n/(n-1)} \; L(n-1)^{n/(n-1)}.$$

Expanding this inequality by repeated substitution on the right hand side yields:

$$L(n) \geq (\prod_{i=2}^{n} i^{n/i-1}) L(1)$$

Set $\mu = r_1 + r_2 + \ldots + r_n$. Recalling that $2^{r_i} \geq \lambda_{i,n}$ for each i, we have

$$2^{\mu} \geq \prod_{i=1}^{n} \lambda_{i,n} \geq |\Lambda_n| \geq L(n)$$

and thus $\mu \geq \log L(n)$, where the base of the logarithm is 2. Then

$$\begin{aligned}
\mu &\geq \sum_{i=2}^{n} \frac{n}{i-1} \log(i) \\
&\geq n \sum_{j=1}^{n-1} \frac{\log j}{j} \\
&\geq n \int_{3}^{n-1} \frac{\log x}{x}\, dx \\
&\geq \frac{n}{2 \ln 2} \left(\ln^2 (n-1) - \ln^2 3 \right).
\end{aligned}$$

Here the third inequality follows from the fact that the function $f(x) = \frac{\log x}{x}$ is decreasing for $x > e$.

We conclude that $\mu = r_1 + r_2 + \ldots + r_n$ is in $\Omega(n \log^2 n)$. $\Box$

4 The Upper bound

In our scheme we represent a permutation by the *successor* relation. This relation consists of the set of pairs (a, b) such that b is located *exactly* after a in the permutation. Each such pair needs $2 \log n$ storage bits. Our goal is to maintain the set of $n-1$ successor pairs so that the permutation can be reconstructed at any moment. We will show that this can be achieved by having $3 \log n$ pairs in the register of each processor. The total requirement is $O(\log^2 n)$ space per processor.

For simplicity we assume that n is a power of 2; minor alterations are required for the general case. For $k \leq n$, consider the abstract data type $M_{n,k}$ which is the same as move-to-front list except that the report operation is replaced by weaker operation $report_k$ that returns only the first k elements in order. Thus $M_{n,n}$ is a move-to-front list.

The construction of an implementation of $M_{n,n}$ consists of a sequence of $1 + \log n$ levels. Level 0 consists of a simple implementation of $M_{n,1}$ with register size $\log n$. For $b \geq 1$, level b of the construction consists of an implementation of $M_{n,2^b}$ that uses an implementation of $M_{n,2^{b-1}}$. The implementation of each level requires an additional $6 \log n$ bits per register. Summing over levels we obtain:

Theorem 4.1 *For each $k \geq 0$, there is an implementation of move-to-front list for $n = 2^k$ processors where each processor's register has size $6 \log^2 n + \log n$ bits.*

Each processor's register is partitioned into segments indexed by b between 0 and $\log n$ where the bth segment stores the information for the implementation of $M_{n,2^b}$ data object.

When describing a particular level of the construction we restrict our attention to the part of the register devoted to that level.

First we give an easy algorithm to implement the $M_{n,1}$ due to Li and Vitanyi [LV89]. In this case all we have to do is to maintain the identity of the first processor in the list.

Lemma 4.2 *$M_{n,1}$ can be implemented with single-writer multiple-reader registers of size* $\log n$.

Proof. Let t_r be the part of the register of processor r devoted to the implementation of $M_{n,1}$. The ID of the first processor in the list is given by $1 + (\sum_{j=1}^{n} t_j) \bmod n$. In a move-to-front(r) operation, processor r assigns t_i a value such that

$$1 + (\sum_{j=1}^{n} t_j) \bmod n = r$$

namely

$$t_r = (r - 1 - \sum_{j=1,\ j\neq r}^{n} t_j) \bmod n$$

Observe that t_r need not have more than $\log n$ bits. □

The scheme can be modified slightly by storing the ID of each processor in its label, so that the outcome is a function of the labels only. For this model the above scheme uses n^2 different labels. Cori and Sopena [CS91] give a better construction with $2n-1$ labels.

For the remainder of the section we describe the implementation of the abstract data object $M_{n,2^{b+1}}$ assuming that an implementation of $M_{n,2^b}$ is given.

Let $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ be the current permutation of the processors. By the induction hypothesis we assume that report$_{2^b}$ returns $\sigma_1, \ldots, \sigma_{2^b}$. In the inductive step, we want to construct $M_{n,2^{b+1}}$ so as to maintain the additional $\sigma_{2^b+1}, \ldots, \sigma_{2^{b+1}}$ part of the permutation. For this purpose, it is sufficient to be able to reconstruct the set of ordered pairs $P_{b+1}(\sigma) = \{(\sigma_m, \sigma_{m+1}) : 2^b \leq m < 2^{b+1}\}$.

For each processor r let s_r denote its register segment for this level. In our implementation each s_r stores a set of pairs (a, b) where a and b are processors; a is referred to as the head of the pair and b as the tail. The heads of all pairs in a given s_r will be distinct. When a processor performs a move-to-front operation, it will change the set of pairs it stores in a manner to be specified. The idea is to make it possible to reconstruct the set $P_{b+1}(\sigma)$ from the pairs stored in the set of all register segments s_r. The difficulty is that, since each processor's register is only updated when it performs a move-to-front, some of the pairs stored will be invalid and we must distinguish between the currently valid pairs and the invalid ones. The information available to the algorithm will then be the sequence of $\sigma_1, \ldots, \sigma_{2^b}$ returned by report$_{2^b}$ together with the sequence $\mathcal{A} = A_1, \ldots, A_{2^b}$ where A_i is the set of pairs in s_{σ_i}. The tuple $C = \{\sigma_1, \ldots, \sigma_{2^b}; \mathcal{A}\}$ is called the *configuration* of the data structure (with respect to level $b+1$)).

Define $B_j(C)$ to be the set of pairs belonging to A_j whose head is not the head of any pair in A_i for $i < j$. Thus all pairs appearing in $B_1(C), \ldots, B_{2^b}(C)$ have different heads. We will maintain the following invariant.

I(a). Every pair in $P_{b+1}(\sigma)$ occurs in one of the $B_i(C)$ sets with $1 \leq i \leq 2^b - 1$.

As we now describe, this invariant is enough to reconstruct the permutation.

THE ALGORITHM FOR REPORT$_{2^{b+1}}$: First report$_{2^b}$ is called and returns $\sigma_1, \ldots, \sigma_{2^b}$. The registers of these processors determine $B_1(C), \ldots, B_{2^b-1}(C)$. Starting from σ_{2^b}, each pair (σ_m, σ_{m+1}) for $2^b \leq m < 2^{b+1}$ is determined to be the (unique) pair from $\bigcup_{1 \leq i < 2^b} B_i(C)$ with head σ_m. The existence of these pairs is guaranteed by the invariant I(a).

For each i with $1 \leq i < 2^b$, define $E_i(C)$ to be the intersection of $B_i(C)$ and $P_{b+1}(\sigma)$. These are the pairs actually used by the algorithm to reconstruct the sequence $\sigma_{2^b+1}, \ldots, \sigma_{2^{b+1}}$. Invariant **I(a)** implies that these sets partition $P_{b+1}(\sigma)$. The pairs belonging to $E_i(C)$ are said to have *height* i.

We must show how to implement move-to-front$_{2^{b+1}}(r)$ by processor r in such a way that invariant **I(a)** is maintained. Let σ and τ denote the permutations before and after the operation is performed. Let p be the index such that $\sigma_p = r$. Obviously $\tau_1 = \sigma_p, \tau_i = \sigma_{i-1}$ for $2 \leq i \leq p$ and $\tau_i = \sigma_i$ for $i > p$.

Let C denote the configuration prior to the move-to-front operation. We need to specify the set of pairs that processor σ_p should write in its register so that the resulting new configuration D satisfies **I(a)**, that is, the sets $B_i(D)$ for $1 \leq i < 2^b$ should contain all of $P_{b+1}(\tau)$. It will be useful to partition the pairs of $P_{b+1}(\tau)$ into sets as follows:

- N: the set of *new* pairs, $P_{b+1}(\tau) - P_{b+1}(\sigma)$. These are pairs that processor r *must* write in its register.
- R: the set of *retained* pairs, consisting of those pairs of $E_p(C)$ which are in $P_{b+1}(\tau)$. These are pairs that already exist in processor r's register and must remain there. Note that R is empty if $p \geq 2^b$.
- T: the set of *threatened* pairs, which is defined to be empty if $p < 2^b$ and $E_{2^b-1}(C) \bigcap P_{b+1}(\tau)$, if $p \geq 2^b$. In words, T consists of those pairs which are currently stored by the processor at position $2^b - 1$ in the list in the case that this processor will be pushed to position 2^b after the operation. Therefore to maintain invariant **I(a)**, processor r must copy the pairs of T.
- S: the set of *safe* pairs, consisting of all other pairs.

It is now easy to see that to maintain invariant **I(a)**, it is sufficient for processor p to record all pairs in $N \bigcup R \bigcup T$. This rule will give a correct implementation for level $b+1$ of the data structure. The problem with this rule is that it does not give any control on the size the set $N \bigcup R \bigcup T$, and in fact allows the set to grow to be as large as some constant fraction of n. This would then require very large registers. To limit register size, we want to maintain another invariant.

I(b). The number of pairs in each processor's register segment is at most 3.

For $b = 0$ or $b = 1$ this invariant is trivially true since $|P_{b+1}(\tau)| < 3$. For $b \geq 2$, we will achieve this invariant by requiring that, in some cases, the processor performing the move-to-front record a single pair from S in addition to all of the pairs in $N \bigcup R \bigcup T$. It is easy to see that this will not affect the correctness of invariant **I(a)**, and we will show that this will guarantee invariant **I(b)**. Our analysis will require that we maintain yet another invariant. For a configuration C, let S_m^C, $1 \leq m \leq 2^b$ be the number of pairs of $P_{b+1}(\sigma)$ of height less or equal to m.

I(c). $S_m^C \geq m + 1$ for $1 \leq m < 2^b$ for level $b \geq 2$.

This invariant asserts that the pairs of $P_{b+1}(\sigma)$ tend to be concentrated in the processors in the beginning of the permutation and are at worst uniformly distributed over the first $2^b - 1$ processors.

THE ALGORITHM FOR MOVE-TO-FRONT$_{2^{b+1}}(r)$: First processor r performs report$_{2^{b+1}}$ to determine $\sigma_1, \ldots \sigma_{2^{b+1}}$ from which $\tau_1, \ldots \tau_{2^{b+1}}$ and $P_{b+1}(\tau)$ can be deduced. The configuration C and the permutation τ determine the partition N, T, R, S of $P_{b+1}(\tau)$. Now processor r rewrites its register to contain the pairs of $N \bigcup R \bigcup T$. In addition, if $R \bigcup T$ is empty and S is not empty, the processor writes a pair (a, b) of S of maximum height in the configuration C. Finally it performs move-to-front$_{2^b}(r)$ to update the lower level data structures.

Since the processor records all pairs in $N \bigcup R \bigcup T$ and possibly one pair in S, it is easy to see that invariant **I(a)** holds for the permutation τ in the new configuration D. We now show that invariants **I(b)** and **I(c)** are also maintained.

Theorem 4.3 *Assume that invariants* **I(a)**,**I(b)**, *and* **I(c)** *hold for some configuration C, and let D be the configuration after a move-to-front$_{2^{b+1}}(r)$ operation. Then the invariants also hold in configuration D.*

Proof. Invariant **I(b)** follows immediately from the following facts about the sets N, T, R and S.

Lemma 4.4

1. *If $p < 2^b$ then N and T are empty and $|R| \leq 3$.*
2. *If $p \geq 2^b$ then $|N| \leq 2$, $|T| \leq 1$ and R is empty.*

Proof. Let F denote the first 2^b and G denote the next 2^b processors in the permutation $\sigma_1, \sigma_2, \ldots, \sigma_n$. Let F' and G' be the corresponding sets for $\tau_1, \tau_2, \ldots, \tau_n$. Figure 1 is a helpful reference for the following discussion.

1. If $1 \leq p < 2^b$ then the move-to-front operation only rearranges the elements of F_{2^b}. Therefore

$$F' = F, \qquad G' = G$$

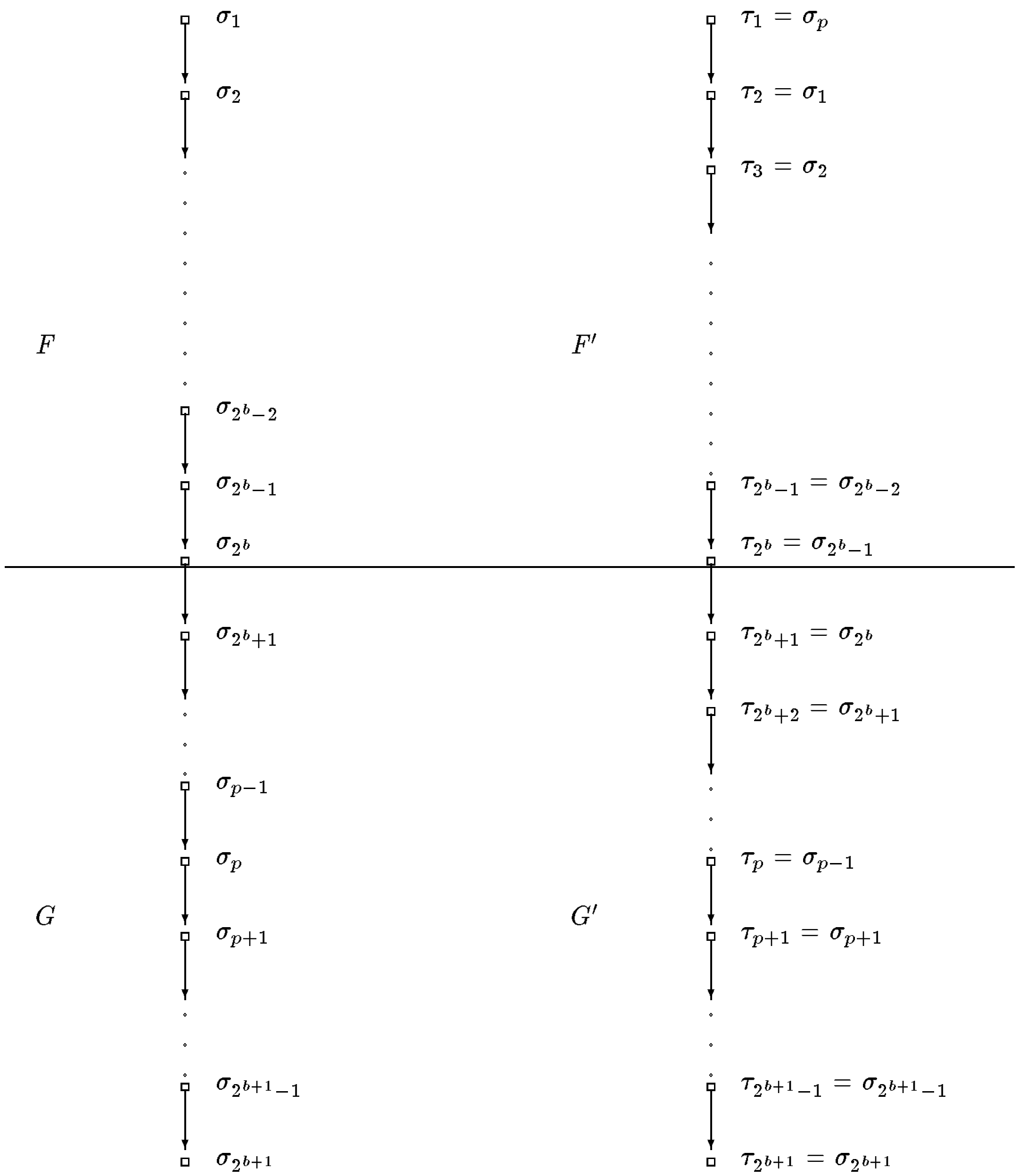

Figure 1: Move-to-front$_{2^{b+1}}(\sigma_p)$ for $p \geq 2^b$

Also $P_{b+1}(\sigma) = P_{b+1}(\tau)$ since the permutation in the next 2^b positions remains the same. Therefore $N = \emptyset$. Furthermore, $T = \emptyset$ by the definition of T and $|R| \leq 3$ because of invariant **I(b)** for C.

2. The cardinality of $P_{b+1}(\sigma)$ is *exactly* 2^b for any permutation σ. Because of the invariant **I(c)**, $S^C_{2^b-2} \geq 2^b - 1$. Therefore σ_{2^b-1} has at most one pair of $P_{b+1}(\sigma)$ and $|T| \leq 1$. Since $p \geq 2^b$, $E_p(C) = \emptyset$ and therefore $R = \emptyset$.

Note that if $2^b \leq p \leq 2^{b+1}$ then

$$\begin{aligned} F' &= F \cup \{\sigma_p\} - \{\sigma_{2^b}\} \\ G' &= G \cup \{\sigma_{2^b}\} - \{\sigma_p\} \end{aligned}$$

and if $2^{b+1} < p \leq n$ then

$$\begin{aligned} F' &= F \cup \{\sigma_p\} - \{\sigma_{2^b}\} \\ G' &= G \cup \{\sigma_{2^b}\} - \{\sigma_{2^{b+1}}\} \end{aligned}$$

Therefore the only pairs in $N = P_{b+1}(\tau) - P_{b+1}(\sigma)$ are $(\sigma_{2^b-1}, \sigma_{2^b}) = (\tau_{2^b}, \tau_{2^b+1})$ and in the case $2^b < p \leq 2^{b+1}$, the pair $(\sigma_{p-1}, \sigma_{p+1}) = (\tau_p, \tau_{p+1})$. Thus $|N| \leq 2$. □

It is worth emphasizing the importance of invariant **I(c)** in the above proof. By guaranteeing that the pairs of P_{b+1} are at worst uniformly distributed among the registers of the first $2^b - 1$ processors, it ensures that the set T has at most one element.

To complete the proof we have to show that **I(c)** is maintained after a move-to-front operation. This is an immediate consequence of the following lemma.

Lemma 4.5 *For each m such that $1 \leq m < 2^b$ one of the following two condition hold.*

$$S^D_m \geq S^C_{m-1} + 1 \tag{1}$$

$$S^D_m \geq S^C_m \tag{2}$$

Proof. Define the index θ to be the maximum height with respect to the configuration C among all pairs in $S \bigcup T$. Note that if T is nonempty then $\theta = 2^b - 1$. The proof is a case analysis according to the relative order of m, p and θ.

Case 1. $m \leq \min\{p, \theta\}$. Move-to-front(r) caused the first $m - 1$ processors to shift by one in the permutation. The number of pairs of $P_{b+1}(\tau)$ stored by these processors is at least $S^C_{m-1} - |P_{b+1}(\tau) - P_{b+1}(\sigma)| = S^C_{m-1} - |N|$. Processor r writes all the pairs in N and at least one other pair, either because $R \bigcup T$ is not empty or because it copies a pair from S of height θ. Therefore $S^D_m \geq (S^C_{m-1} - |N|) + (|N| + 1)$ and inequality (1) follows.

Case 2. $m > \theta$. Observe that in configuration D, all pairs of $P_{b+1}(\tau)$ are of height at most $\theta + 1$. Therefore $S^D_m = 2^b \geq S^C_m$.

Case 3. $p < m \leq \theta$. Since $p \leq m < 2^b$ then $P_{b+1}(\sigma) = P_{b+1}(\tau)$, and the two sets of processors $\{\sigma_1, \ldots, \sigma_m\}$ and $\{\tau_1, \ldots, \tau_m\}$ are equal. Thus any pair of height at most m in the configuration C also has height at most m in configuration D and inequality (2) follows. □

	π_{p-1}			π_{p-2}	$\cdots$	$\cdots$	π_1	
π_p								
		π_{p+1}	π_{p+2}		$\cdots$	$\cdots$		π_n
σ_1	σ_2	σ_3	σ_4	σ_5	$\cdots$	$\cdots$	σ_{n-1}	σ_n

Table 1: An example of the relation between permutations π and σ

5 Move-to-p lists

The move-to-front list supports movement of an arbitrary processor to the front of the list. An obvious generalization of the move-to-front list would be a *generalized list* that allows a processor to move from any position in the list to any other position in the list.

Question. Is there an implementation of a generalized list, in which all processors have register size polylogarithmic in n?

We do not have an answer to the above question. In this section though, we answer an easier question by showing that a simple modification of the move-to-front list can implement a list which supports the movement of any processor to position p on the list, where p is an arbitrary *but fixed* integer in the range 1 to n. Such a data structure is called a move-to-p list and supports two operations:

- *report :* returns the permutation
- *move-to-p(i) :* moves element i to position p of the permutation.

Move-to-front list corresponds to $p = 1$. We examine this family of data objects under the assumptions $1 - 4$ presented in the case of the move-to-front list.

The main part of the implementation of the move-to-p list is to maintain an associated move-to-front list, called the *ghost* list. This list is initialized and updated as follows. Let π^0 denote the initial permutation of processors. The initial ghost list σ^0 is defined from π^0 by reversing the order of the first p processors. The ghost list is updated according to the following rule: each time a processor r performs a move-to-p action, the processor r moves to the front of the ghost list. Thus the ghost list orders the processors according to when they most recently performed a move-to-p operation. Trivially we can maintain the ghost list using the move-to-front implementation already defined. In what follows, we use π to denote the "current" order of the move-to-p list that is being maintained and σ to denote the associated ghost list.

Now, if it were the case that the list π is always uniquely determined by the ghost list σ then maintaining σ would suffice. However, this is easily seen not to be the case. The question is: what additional information is needed to reconstruct the true list from the ghost list? For the permutation π, let $F = F(\pi)$ denote the set of indices that appear in the first $p-1$ positions of π and $L = L(\pi)$ denote the set of indices that appear in the last $n-p$ positions, so the sets $F(\pi)$ and $L(\pi)$ partition $\{1, 2, \ldots, n\} - \{\pi_p\}$.

The following lemma asserts that σ together with the two sets $F(\pi)$ and $L(\pi)$ uniquely determine π. Furthermore it specifies how π can be reconstructed.

Lemma 5.1 *Let π be the list produced by a sequence of move-to-p operations and σ be the associated ghost list. Then:*

1. *The elements of $L(\pi)$ occur in exactly the same order in π as they do in σ.*
2. *The elements of $F(\pi)$ occur in exactly the same order in π as they do in permutation σ^R obtained by reversing σ.*

In particular, π is uniquely determined by σ and the two sets $F(\pi)$ and $L(\pi)$.

Table 1 presents an example of the relation between the two permutations.

Proof. The proof is by induction on the number of move-to-p operations. The assertion is true for π^0 by the definition of σ^0. For the induction step, assume that the conclusion of the lemma holds for a given π and associated σ, and let π' and σ' be the modified lists after applying the operation move-to-$p(i)$ for some processor i. Clearly, if i was already in position p of π, and therefore in the first position of σ, then the operation leaves both π and σ unchanged. Let $i = \pi_r$ where $r \neq p$. We distinguish two cases.

Case 1. $r < p$.

Then $L(\pi) = L(\pi')$ and, in addition, the relative order of the set $L(\pi)$ is the same in π' as in π and is the same in σ' as in σ. Since $L(\pi)$ has the same ordering in σ and π, by the induction hypothesis, the same is true for σ' and π'.

Next note that in this case, $F(\pi') = F(\pi) - \{\pi_r\} \cup \{\pi_p\}$. Now the relative order of all elements of the set $F(\pi') - \{\pi_p\} = F(\pi) - \{\pi_r\}$ is the same in π,π',σ^R, and σ'^R. Furthermore, $\pi_p = \sigma_1$ becomes $\pi'_{p-1} = \sigma'_2$, so that π_p is last among $F(\pi')$ in both π' and σ'^R. Thus $F(\pi')$ appears in the same order in π' and σ'^R.

Case 2. $r > p$.

In this case, $F(\pi) = F(\pi')$. Furthermore, since neither π_p nor π_r belong to $F(\pi')$, the relative order of the set $F(\pi')$ is the same in π' as in π and is the same in σ'^R as in σ^R. Since $F(\pi)$ has the same ordering in σ^R and π, by the induction hypothesis, the same is true for σ'^R and π'.

Next observe that in this case, $L(\pi') = L(\pi) - \{\pi_r\} \cup \{\pi_p\}$. Now the relative order of all elements of the set $L(\pi') - \{\pi_p\} = L(\pi) - \{\pi_r\}$ is the same in π,π',σ, and σ'. Furthermore, $\pi_p = \sigma_1$ becomes $\pi'_{p+1} = \sigma'_2$ so that π_p is first among $L(\pi')$ in both π' and σ'. Thus $L(\pi')$ appears in the same order in π' and σ'. □

In light of this lemma, it suffices to augment the data structure for maintaining the ghost list, so that it also maintains the two sets $F(\pi)$ and $L(\pi)$. This is done by modifying the move-to-front implementation as follows. As described in the previous section, the basic unit of data in the move-to-front implementation is a pair (a,b) of processors. In the new implementation, each such pair (a,b) in a processor's register is labeled by a *tag* variable, $\beta \in \{L,F\}$. The idea is that β is supposed to specify the set to which processor b belongs. Thus, the tags in the register of an arbitrary processor i will contain a view of the membership of certain other processors in L and F. This view will be correct at the time of i's last move-to-p operation, when it last rewrote the register. As before, we face the problem of distinguishing current information about membership from outdated information.

To perform a *report* operation, a processor uses the move-to-front *report* operation to reconstruct σ. Then $\pi_p = \sigma_1$. Each processor $b \neq \pi_p$ is assigned to L or F according to the tag for b that appears in the register of the first processor in the σ order whose register contains such a tag. ¿From σ, L and F, π can be reconstructed as described above.

To perform a *move-to-p* operation, a processor first performs *report* operation to determine the present σ ordering and the sets L and F. It then performs a move-to-front operation on σ, assigning the correct tag to all pairs that writes in its register.

We sketch the proof of correctness of the above modification, leaving the details to the reader. The main step is to show that, in a *report* operation, each processor r is assigned to the correct set L or F. Consider a permutation π and the associated ghost σ and suppose that $r = \sigma_j$. For the given implementation of the move-to-front list, we know that at least one processor s that precedes r in σ order must have the pair (σ_{j-1}, r) stored in its register and thus must have a tag for r. To see that this tag is correct , it suffices to observe: (1) The set to which r belongs was determined at the time of the last move-to-p operation immediately following r's last move-to-p operation, or by the initialization if r never performed a move-to-p operation, and (2) all processors that precede r in σ have updated their registers since r's last move-to-p operation and thus anyone that has a tag for r must have a correct tag for r.

To implement the move-to-$p(r)$ operation, we need to ensure that in the ghost list, we can maintain tags that correctly specify F and L. If a more-to-$p(r)$ operation is applied when the permutation is π and the ghost permutation is σ then the only element added to either L or F is π_p. Since r moves to the front of the new ghost list σ' and π_p is second in that list, r will have in its register the pair (r, π_p) and can affix the appropriate tag to that pair.

Acknowledgements: We benefited greatly from discussions with Robert Cori. The second author also wishes to thank Elias Koutsoupias for useful discussions.

References

[AK+91] Y. Azar, A. Karlin, J. Naor and N. Shavit. *Personal Communication* 1991.

[CS91] R. Cori, E. Sopena. *Some Combinatorial Aspects of Time Stamp Systems.* Unpublished manuscript 1991.

[DS89] D. Dolev and N. Shavit. *Bounded Concurrent Time-Stamp Systems are Constructible!* Proceedings of the 21^{th} Annual ACM Symposium on Theory of Computing, 1989, pp. 454-465. Also Technical Report, IBM, March 1990.

[DW92] C. Dwork and O. Waarts. *Simple and Efficient Bounded Concurrent Timestamping or Bounded Concurrent Timestamp Systems are Comprehensible!* Proceedings of the 24^{th} Annual ACM Symposium on Theory of Computing, 1992, pp. 655-666.

[IL87] A. Israeli and Ming Li. *Bounded Time-Stamps.* Proceedings of the 28^{st} Annual Symposium on Foundations of Computer Science, October 1987 pp. 371-382.

[La78] L. Lamport : *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM, 21(7) July 1978, pp. 558-565.

[LV89] M. Li and P. Vitanyi. *How to Share Concurrent Asynchronous Wait-Free Variables.* Proceedings of 16^{th} ICALP 1989, pp. 488-505.

[SZ91] M. Saks and F. Zaharoglou. *Optimal Space Distributed Move-To-Front Lists.* Proceedings of the 10^{th} ACM Symposium on Principles of Distributed Computing, August 1991, pp. 65-73.

[Za93] F. Zaharoglou. *Distributed Data Structures and Wait-Free Computation.* Ph.D. Thesis, University of California, San Diego, September 1993.