The Scheduler is Very Powerful in Competitive Analysis of Distributed List Accessing *

Joan Boyar\textsuperscript{1} Faith Ellen\textsuperscript{2} Kim S. Larsen\textsuperscript{1}

\textsuperscript{1} University of Southern Denmark, \{joan,kslarsen\}@imada.sdu.dk
\textsuperscript{2} University of Toronto, faith@cs.toronto.edu

Abstract

This work is a continuation of efforts to define and understand competitive analysis of algorithms in a distributed shared memory setting, which is surprisingly different from the classical online setting. In fact, in a distributed shared memory setting, we find a counter-example to the theorem concerning classical randomized online algorithms which shows that, if there is a \(c\)-competitive randomized algorithm against an adaptive offline adversary, then there is a \(c\)-competitive deterministic algorithm \[18\]. In a distributed setting, there is additional lack of knowledge concerning what the other processes have done. There is also additional power for the adversary, having control of the scheduler which decides when each process is allowed to take steps.

We consider the list accessing problem, which is a benchmark problem for sequential online algorithms. In the distributed version of this problem, each process has its own finite sequence of requests to a shared list. The scheduler arises as a major issue in its competitive analysis. We introduce two different adversaries, which differ in how they are allowed to schedule processes, and use them to perform competitive analysis of distributed list accessing. We prove tight upper and lower bounds on combinatorial properties of merges of the request sequences, which we use in the analysis. Our analysis shows that the effects of the adversarial scheduler can be quite significant, dominating the usual quality loss due to lack of information about the future.

1 Introduction

Our aim is to improve our understanding of competitive analysis of algorithms in a distributed shared memory setting. We investigate a theoretical benchmark online problem, List Accessing. New problems arise due to scheduling, when the processes each have their own finite request sequences. These problems are not addressed in existing models for competitive analysis in distributed settings, so we introduce two new adversarial models. Through a sequence of results, we expose which circumstances affect worst-case performance the most. In standard online algorithms, the lack of knowledge of the future is often the primary obstruction, but in the distributed setting, the scheduling of events seems to play an even larger role.

\*Supported in part by the Independent Research Fund Denmark, Natural Sciences, grant DFF-7014-00041 and the Natural Science and Engineering Research Council of Canada.
1.1 Online Algorithms and Competitive Analysis

A problem is called online if the input is given one piece at a time and an algorithm must make an irrevocable decision regarding each piece before the next piece is given. The pieces of the input are called requests. An online algorithm is an algorithm for solving an online problem. The goal of an online algorithm is to minimize or maximize an objective function. In this paper, we restrict the discussion to minimization problems and call the objective function the cost.

The most common technique for analyzing online algorithms is competitive analysis \cite{competitive_analysis}. If Alg(I) denotes the cost of running the algorithm Alg on the input sequence I, then Alg is said to be c-competitive, if there exists a constant \( \alpha \) such that for all I, \( \text{Alg}(I) \leq c \text{OPT}(I) + \alpha \), where OPT is a (hypothetical) offline, optimal algorithm. The competitive ratio of an algorithm is the infimum over all c for which it is c-competitive.

1.2 Competitive Analysis for Distributed Algorithms

We consider a standard model of asynchronous, shared memory computation \cite{shared_memory_computation} in which \( p \) processes communicate with one another by performing read, write, and compare-and-swap (CAS) on shared memory locations. CAS(\( x, \text{old}, \text{new} \)) atomically returns the value of \( x \) and, if it is old, changes it to new. In this model, processes may run at arbitrarily varying speeds and may even crash. Each shared memory access is one step, and the goal, after correctness, is to minimize the number of steps. An execution is modeled as a sequence of input events, output events, and shared memory accesses starting from an initial configuration. The schedule (that is, the order in which processes take steps) and what inputs they are given are viewed as being under the control of an adversary.

Overall, the purpose of applying competitive analysis (comparing to some optimal algorithm, OPT) is to give a realistic idea of how much improvement of an algorithm might be possible, as opposed to simply using a worst-case analysis. Its goal is to measure the excess cost of a distributed algorithm as compared to the cost of an optimal algorithm in related situations.

In studying competitive analysis for distributed algorithms, we first consider the well-known theorem from classic online algorithms which says that if there is a c-competitive randomized algorithm against an adaptive, offline adversary, then there is a c-competitive deterministic algorithm \cite{randomized_algorithm}. We present a counter-example, showing that this is not true in a distributed setting.

**Theorem 1** There exists a problem, FINDVALUE, in a distributed setting, where there is a randomized algorithm which is \( \frac{23}{16} \)-competitive against an adaptive, offline adversary, while the best deterministic algorithm is no better than \( \frac{3}{2} \)-competitive.

The first papers applying competitive analysis in a distributed setting \cite{distributed_competition_1, distributed_competition_2} applied it in a message passing, rather than a shared memory setting. They compared the cost of a distributed online algorithm to the cost of an optimal offline, sequential algorithm. Even so, they achieved good competitive ratios for a job scheduling problem and a data management problem. This same definition of competitive analysis is used in \cite{distributed_competition_3, distributed_competition_4, distributed_competition_5, distributed_competition_6} and elsewhere, including \cite{distributed_competition_7} with extra resource analysis. In \cite{lower_bound}, a lower bound is proven in this model, explicitly assuming that the request sequence is given sequentially. In these papers, OPT has global control and does not pay for the overhead of learning the relevant part of the global state to make its decisions. Aspnes and Waarts \cite{lower_bound} have argued that this might be fair for problems where the purpose is to manage
resources, but unfair for problems where the main purpose is to propagate information. The first paper to propose a model of competitive analysis without global control was [1].

Online algorithms are faced with uncertainty as a result of lack of knowledge about future requests: irrevocable decisions regarding early requests may be unfortunate when future requests arrive. In his survey on the competitive analysis of distributed algorithms [8], Aspnes observed that distributed online algorithms also face uncertainty about the scheduler. Actions of the processes may be scheduled in a beneficial or counter-productive manner with regards to the overall objective function. Aspnes compares algorithms to OPT on the same schedule. He considers comparing them on the same input, on the worst-case input for the algorithm and the best-case input for OPT, or on the worst-case input for the algorithm and the worst-case input for OPT. He considers the repeated collect problem in a shared memory model and presents the proof that the randomized Follow-the-Bodies algorithm [9] is $O(\log^3 n)$-competitive.

In [10], the focus is modularity, trying to make competitive analysis compositional. This necessitates that algorithms and OPT may be compared on different inputs. Alistarh et al. [5] compare an asynchronous algorithm to OPT on a worst schedule for OPT.

Online algorithms have been considered for multi-threaded problems, including paging [25], multi-sequence prefetching [28], and multi-threaded metrical task systems [26]. In these problems, the input consists of $p$ lists of requests, but requests are treated by one server, who can choose which request to treat next from among the first outstanding requests in each list. The server being analyzed is not distributed.

1.3 The Benchmark Problem: List Accessing

The (static) List Accessing problem is a theoretical benchmark problem. In fact, it was one of the original two problems studied with competitive analysis and the first with amortized analysis [31]. In addition, it is the first major example and a means of introducing randomized online algorithms in the standard textbook [20]. It has been used to explore performance measure issues using lookahead [30], bijective analysis [7], parameterized analysis [24], relative worst-order analysis [24], and advice complexity [22, 21], as well as issues related to self-adjusting data structures.

In this problem, we must administrate a linked list of $\ell$ items. A request is an item in the list, and the algorithm must find that item in the list, each time starting from the front of the list. When the item has been found, the algorithm may move it to any location closer to the front of the list. In the sequential online setting, the (possible) move must be completed before the next request is given to the algorithm. The cost is the total number of item accesses during the search. Thus, an algorithm for this problem is simply a strategy, detailing when an item should be moved and where it should be placed. (Paid exchanges, where an algorithm is charged one for switching the order of two adjacent items, are also allowed, but we only consider them indirectly when known results are used.)

According to competitive analysis, the well-known Move-to-Front algorithm (MTF), which always moves an accessed item to the front of the list, is an optimal deterministic algorithm in the sequential setting. MTF has (strict) competitive ratio $2 - \frac{2}{\ell + 1}$ [27]. (Referring to personal communication, Irani credits Karp and Raghavan with the lower bound.)
1.4 Competitive Analysis of Distributed List Accessing

For our distributed version of List Accessing, the list, of fixed length $\ell$, is shared among $p$ processes, each of which has its own request sequence. Each process must access every item in its own request sequence in order (unless it fails and stops), always starting at the front of the list for each search. The cost of a search is the total number of shared memory accesses during the search. In the worst case, this is asymptotically proportional to the total number of items accessed, which, for each request, is the index of the item, starting from the front of the list.

Our algorithm is a distributed version of MTF, so each search for an item is followed by a move-to-front operation, unless the item is already at the front of the list. We are not assuming that OPT performs a move-to-front after every request. Even though OPT knows all sequences, it must respect the order of each process’s requests sequence: Only after it has finished a request, can it proceed to the next request of the same process, initiating this search from the beginning of the current shared list.

If the online algorithm and OPT can be given different request sequences, as in [10], it is easy to show too pessimistic a lower bound for List Accessing. For example, the algorithm gets requests to the end of the list, whereas OPT gets requests to the beginning of the list. Thus, in our models, the $p$ processes are given exactly the same request sequences in the online algorithm as in OPT.

Although OPT with global control is considered too pessimistic for problems concerned with propagating information [10], this is not the concern of the List Accessing problem. Contention among the processes only hurts the performance in the worst case, because the same work may be done by multiple processes. In both of our models, OPT has global control: it knows the entire schedule in advance. As with earlier work that considers an optimal algorithm, OPT, with global control, we also assume that OPT is sequential. Theorem [16] shows that giving OPT global control only increases the competitive ratio by a factor of 2, which is the factor that comes from the standard online competitive analysis of MTF. In contrast, with our fully adversarial scheduler defined below, a factor of $2p^2 - p$ comes from the power of the scheduler, which is assumed to give a best possible schedule to OPT and a worst possible schedule to the distributed MTF algorithm. Consider the following simple example: Suppose the shared list starts as $L = [x_1, x_2, \ldots, x_\ell]$ and the $p$ processes each have the same request sequence, $\sigma = \langle x_\ell, x_{\ell-1}, \ldots, x_1 \rangle$, repeated $s$ times. If the scheduler arranges that each process makes all of its requests before the next begins, then each item is at the end of the list when it is requested and has cost $\ell$. If the scheduler arranges that the processes take steps in a round robin fashion, then all $p$ processes can be looking for the same last item in the list at the same time, each checking if it is the first item in the list before any checks if it is the second, etc. Again, all processes have cost $\ell$ for finding each item. (The problem associated with all trying to perform the move-to-front simultaneously is an additional problem, discussed later.) However, if each process finishes the search and move-to-front for their item before the next begins, all but the first would only have cost 1.

Even if OPT and the distributed MTF algorithm are given the same schedule, there does not seem to be any way to use this information. Since OPT may complete some operations in fewer steps than the distributed MTF algorithm, they could eventually be searching for different items at the same point in the schedule and the lists in which they are searching may be different. Thus, we do not assume the same schedule for the distributed algorithm and for OPT.
1.5 New Adversarial Models

We consider a distributed version of MTF, which we call DMTF. A specific lock-free implementation is presented in Section 6. Most of our competitive analysis of DMTF does not depend on the specific implementation.

We assume there are \( p \) processes and each process has its own input sequence. We analyze DMTF using two different models. In both, we assume DMTF and OPT are processing the same \( p \) input sequences.

First, we allow OPT to have a completely different schedule than DMTF. We call this the **fully adversarial scheduler** model. With completely different schedules, one can only be certain that each process performs its requests in the order specified by its request sequence. However, requests from other request sequences can be merged into that sequence in any way. This method appears to be closest to the worst-case spirit generally associated with competitive analysis. As far as we know, this is the first time the merging of processes’ request sequences has been considered in competitive analysis of a distributed algorithm, where the merging is not fully or partly under the control of the algorithm being analyzed.

There are three parts to this analysis. First, we only consider the effect of merging the request sequences, requiring operations to be performed sequentially. Next, we consider processes that can be looking for the same item at the same time. Then, we take into account arbitrary contention among processes. The first two parts are independent of the specific implementation of distributed MTF.

- Since the adversary can schedule DMTF and OPT differently, we bound the ratio of costs associated with two different merges of the processes’ request sequences. This corresponds to the competitiveness that can be experienced in a model where interleaving takes place at the operation level, i.e., a search for an item and the subsequent move-to-front is carried out before the next search is initiated. Thus, the execution is sequential after the merging of the request sequences. In this level, the cost of DMTF is MTF’s cost on the merged sequence.

- Then we consider interleaving at the item access level. Here, we take into account that more than one process may be searching for the same item at the same time. However, we assume that all such processes find the item in the same location. Then the item is moved to the front of the list. Again, we only consider the costs of searching through the list, counting cost \( i \) for an item in location \( i \) of the list.

- Finally, we consider an actual algorithm in a fully distributed model. The overhead involved in a move-to-front operation in a distributed setting is considered at this point. One complication is that a process may move an item to the front of the list after another process has started searching for the same item. Additional overhead is required to ensure that the second process does not search all the way to the end of the list without finding the item.

In the fully adversarial scheduler model, OPT may have too much of an advantage over DMTF, because OPT can arbitrarily merge the sequences of requests given to the processes. To understand the effect of this, we consider a second model, which is more similar to the situation in the sequential setting, where OPT performs the same sequence of operations (searches) as MTF. In this model, which we call **linearization-based**, OPT still performs its operations sequentially, but is required to perform them in an order which is a linearization of the execution performed by DMTF. This means that, if operation \( o_1 \) is completed by DMTF before it begins operation \( o_2 \), then OPT must
perform $o_1$ before $o_2$.

The processes that are successful in satisfying a request do so either by

1. finding the item and moving it to the front (if it is not already there),
2. finding the item and discovering that another process has moved it to the front, or
3. being informed about the item by another process.

In all three cases, the item is moved to the front by some process and the concurrent requests to that item are linearized as successive requests to that item, when that item is at the front of the list. Thus, the ordering of the list during any execution of DMTF is the same as it would be if the sequential MTF algorithm were run on the linearized request sequence.

1.6 Results

First, we consider the well-known theorem from classic online algorithms which says that competitive analysis of randomized algorithms against adaptive, offline adversaries is uninteresting, because these randomized algorithms are no better than deterministic algorithms. In Section 2, we present a counter-example, showing that this is not true in a distributed setting.

The main technical result is a proof in the fully adversarial scheduler model, showing that one merge of $p$ sequences can be at most a factor $2p^2 - p$ more costly for MTF than another merge. It is accompanied by a matching lower bound. This corresponds to interleaving at the operation level. The result implies that, in the fully adversarial scheduler model, the scheduler is so powerful that it can ensure that OPT asymptotically does a factor of $\Theta(p^2)$ better than any sequential algorithm, even an optimal offline algorithm that is forced to run sequentially on this adversarial merge. Thus, randomization cannot help here, although it often helps in distributed settings.

In order to prove this result, we prove a property about a new distance measure defined on merges of $p$ sequences. We believe that this combinatorial result could be of independent interest.

At worst, the adversary can choose the input sequences so that, for two different merges, the cost of MTF can differ by a factor of $2p^2 - p$. Furthermore, for any merge, the cost of MTF is at most a factor of 2 larger than the cost of OPT. Thus, for interleaving at the operation level, the ratio of the cost of MTF to the cost of OPT is at most $4p^2 - 2p$. These results are presented in Section 4.

We consider interleaving at the item access level in Section 5 and show that it does not increase the worst-case ratio. Indeed, there will sometimes be wasted work when processes search for the same item, but not in the worst-case scenarios. The lower bound on the ratio for interleaving at the operation level carries over to interleaving at the item access level.

In the linearization-based model, both OPT and MTF have the same input sequence for interleaving at the operation level, so the classic result from online algorithms gives the ratio $2 - \frac{2}{p+1}$ [27]. For interleaving at the item access level, we prove an upper bound of $p + 1$ on the ratio, as opposed to $4p^2 - 2p$ for the fully adversarial scheduler. This appears in Section 8. A lower bound of $p$ follows from the example where all $p$ processes are always searching for the same item at the end of the list.

In Section 6 we present a lock-free implementation of a distributed version of MTF, which we call DMTF. Its complete analysis in either the fully adversarial model with arbitrary interleaving or
the linearization-based model, including the analysis of the move-to-front operation, increases the cost of a search by a constant factor and an additive $O(p^2)$ term. Thus, in the fully adversarial model, the upper bound on the competitive ratio is $O(p^2)$ and, in the linearization-based model, it is $O(p)$ provided that OPT’s average cost per request is $\Omega(p)$. The lower bounds from the item access level carry over to the level of the actual algorithm.

These results concerning List Accessing are summarized in Table 1. Note that the results from the first two levels are independent of the actual distributed algorithm being used; this is not considered until the third level. Thus, the operation level seems to be the dominant level for the fully adversarial model, and the item access level seems to be the dominant level for the linearization-based model.

| Level       | Model                  |
|-------------|------------------------|
| Operation   | $[2p^2 - p, 4p^2 - 2p]$ | $2 - \frac{2}{\ell + 1}$ |
| Item access | $[2p^2 - p, 4p^2 - 2p]$ | $[p, p + 1]$            |
| Actual algorithm | $\Theta(p^2)$      | $\Theta(p)$              |

Table 1: Bounds on the ratio of DMTF to OPT.

Thus, when using competitive analysis in a distributed setting, there is no dependency on $\ell$, the length of the list, which we assume is much larger than the number of processes. This length would show up in the analysis of the worst case amount of work done if competitive analysis was not used.

2 Competitiveness in a Distributed Setting Is Different

In his survey [8], Aspnes presents a competitive analysis of the randomized Follow-the-Bodies algorithm against an adaptive offline adversary. He writes that it is unknown whether there is a deterministic algorithm that performs this well. In the standard sequential online model, such a deterministic algorithm must exist [18]. However, this is not necessarily the case for distributed algorithms. Processes lack information about the state of the system as a whole and must act based on their limited knowledge of that state.

The theorem in [18] (and [20, Theorem 7.3]) states that, for any problem, if there is a $c$-competitive randomized online algorithm against an adaptive, offline adversary, then there exists a $c$-competitive deterministic online algorithm. We use the rest of this section to show that, for a natural problem in a distributed setting, this theorem does not hold.

We define the setting, show matching upper and lower bounds of $3/2$ on on the deterministic competitive ratio and give a randomized synchronous algorithm which is $\frac{23}{16}$-competitive against an adaptive, offline adversary.

2.1 The Model and Problem

We consider the following problem, FINDVALUE, where there are 3 processes, $p_0$, $p_1$, and $p_2$ in a synchronous distributed system. Each process, $p_i$, has one register, $R_i$. The processes communicate
by writing into and reading from these single-writer registers. In each round, each process can flip some coins and, based on its state and on the outcomes of its coin tosses, it can do nothing, it can write to its register, or it can read the register of some other process.

Consider the following problem. From time to time, an adversary gives a number as input to one of the processes and lets it take a step (in which it appends a pair consisting of its process id and this number to its register. In the next round, the adversary notifies each of the other processes that it has produced a new number (by giving a special notification input), but does not tell them to whom it gave this number. The goal is for each process to write the entire sequence of pairs into their single-writer register. Our complexity measure is the number of register reads that are performed (by the two processes trying to find the number).

Each time the adversary notifies a process that it has produced a new number, OPT will perform one read, from the register of the process that received the number, and append the new pair to its single-writer register. Since it must do this for two processes, it must perform two register reads for each input.

### 2.2 Deterministic Upper Bound

The following deterministic algorithm performs 3 reads per input item. Each process \( p_i \) maintains a list of the pairs it has learned about and the total number of notifications it has received from the adversary. When the adversary gives a number as input to process \( p_i \), it appends a pair consisting of \( i \) and this number to its list and writes its list to \( R_i \). When told that a new number is available, process \( p_i \) reads from \( R_{(i+1) \mod 3} \). If there are more pairs in that register than in its list, \( p_i \) appends the extra pairs to its list and writes its list to \( R_i \). If the length of the list in \( R_{(i+1) \mod 3} \) is smaller than the number of notifications it has received from the adversary, then, in the following round, \( p_i \) reads from register \( R_{(i-1) \mod 3} \), appends the extra pairs in that register to its list, and writes its list to \( R_i \).

When process \( p_k \) directly gets a number as input from the adversary, process \( p_{(k-1) \mod 3} \) will read this number from \( R_k \) in the next round. However, process \( p_{(k+1) \mod 3} \) will read from \( R_{(k+2) \mod 3} \) in that round and then will read from \( R_k \) in the following round. Thus, a total of three reads are performed for each input. Thus, the algorithm is \( 3/2 \)-competitive.

### 2.3 Deterministic Lower Bound

An adversary can force 3 reads per input number. It will not give any process an input until all processes know about all of the previous inputs.

For each process \( p_i \), let \( R_{f_i} \) be the first location that \( p_i \) will read from when informed that the next new number is available. Note that \( f_i \neq i \). Suppose there exist two processes, \( p_i \) and \( p_j \), such that \( f_i = f_j = k \). Without loss of generality, suppose that \( f_k = i \). Then the adversary gives the new input number to \( p_j \). Whichever of \( p_i \) and \( p_k \) goes first performs at least 2 reads, for a total of 3 reads.

Otherwise, no two processes read from the same register first. Without loss of generality, suppose that \( f_0 = 1 \), \( f_1 = 2 \), and \( f_2 = 0 \). Consider the minimum number of rounds after receiving a notification input before any one of these processes reads a register. Suppose \( p_k \) is a register that reads in this round. If the adversary gives the new input number to \( p_{(k-1) \mod 3} \), then \( p_k \) reads
from $R_{(k+1) \mod 3}$ in this round and does not see the new number. Hence it has to perform at least 2 reads. Process $p_{(k+1) \mod 3}$ also has to perform at least one read, for a total of 3. Thus, no deterministic algorithm is better than $\frac{3}{2}$-competitive.

### 2.4 Randomized Upper Bound

Consider the following randomized algorithm for this problem. Each process $p_i$ maintains a list of the pairs it has learned about and the total number of notifications it has received from the adversary. When the adversary gives a number as input to process $p_i$, $p_i$ appends a pair consisting of $i$ and this number to its list and writes its list to $R_i$. Whenever process $p_i$ has been told that a new number is available, it flips two fair coins. Based on the outcome of the first coin, $p_i$ decides whether to read in this round or delay its next read for 2 rounds. The second coin is used to choose which of the registers belonging to the other two processes it will read from. Suppose it chooses to read from register $R_j$. If there are more pairs in $R_j$ than in its list, $p_i$ appends the extra pairs in $R_j$ to its list and writes its list to $R_i$ in the next round. If the length of $R_i$ is smaller than the number of notifications it has received from the adversary, $p_i$ also reads from the register $R_k$ of the other process, and appends the extra pairs in $R_k$ to its list.

Suppose that $p_i$ is given the number as input in round $r$. Let $p_j$ and $p_k$ be the other two processes.

With probability $\frac{1}{4}$, $p_j$ reads from $R_i$ in round $r + 1$ and writes the number to $R_j$ in round $r + 2$. In this case,

- $p_j$ performs 1 read,
- $p_k$ performs 1 read with probability $\frac{3}{4}$, and
- $p_k$ performs 2 reads with probability $\frac{1}{4}$ (if it reads from $R_j$ in round $r + 1$).

With probability $\frac{1}{4}$, $p_j$ reads from $R_k$ in round $r + 3$. In this case,

- $p_j$ performs 2 reads,
- $p_k$ performs 1 read with probability $\frac{1}{2}$, and
- $p_k$ performs 2 reads with probability $\frac{1}{2}$ (depending on whether $p_k$ chooses to read from $R_i$ or $R_j$ first).

With probability $\frac{1}{4}$, $p_j$ reads from $R_k$ in round $r + 1$. In this case,

- $p_j$ performs 2 reads,
- $p_k$ performs 1 read with probability $\frac{1}{2}$, and
- $p_k$ performs 2 reads with probability $\frac{1}{2}$ (depending on whether $p_k$ chooses to read from $R_i$ or $R_j$ first).

With probability $\frac{1}{4}$, $p_j$ reads from $R_k$ in round $r + 3$. In this case,

- with probability $\frac{1}{2}$, $p_k$ first reads from $R_j$ and both processes perform 2 reads,
- with probability $\frac{1}{4}$, $p_k$ reads from $R_i$ in round $r + 1$, so both processes perform 1 read, and
- with probability $\frac{1}{4}$, $p_k$ first reads from $R_i$ in round $r + 3$, so $p_k$ performs 1 read and $p_j$ performs 2 reads.
The expected number of reads for this algorithm is

\[
\frac{1}{4} \times (3/4 \times 2 + \frac{1}{2} \times 3) + \frac{1}{4} \times \left(\frac{1}{3} \times 2 + \frac{1}{2} \times 3\right)
+ \frac{1}{4} \times \left(\frac{1}{2} \times 3 + \frac{1}{2} \times 4\right) + \frac{1}{4} \times \left(\frac{1}{3} \times 4 + \frac{1}{4} \times 2 + \frac{1}{4} \times 3\right)
= \frac{1}{4} \times \left(\frac{9}{4} + \frac{5}{2} + \frac{7}{2} + \frac{13}{4}\right) = \frac{23}{8} < 3.
\]

Thus, there is a randomized algorithm which is \(\frac{23}{16}\)-competitive against an adaptive, offline adversary, giving us the following:

**Theorem 2** There exists a problem, **FindValue**, in a distributed setting, where there is a randomized algorithm which is \(\frac{23}{16}\)-competitive against an adaptive, offline adversary, while the best deterministic algorithm is no better than \(\frac{3}{2}\)-competitive.

This is a counterexample showing that the theorem in [18], making results against adaptive, offline adversaries uninteresting in the sequential online setting, does not necessarily apply to distributed settings.

### 3 MTF, OPT, and the Distance Measure

A **request sequence** is a sequence of (requests to) items from a list of size \(\ell\). A request to the same item can appear multiple times in a request sequence, so we are often working with the index into a sequence. We use \(I, J\), and \(K\) to refer to generic request sequences. For some request sequence \(I\) of length \(n = |I|\), its sequence of requests is denoted \(I_1, I_2, \ldots, I_n\). Our notation is case-sensitive, so \(I_i\) denotes the request with index \(i\) in the request sequence \(I\). We use \(i, j, k, x, y, \text{ and } z\) to denote indices. The concatenation of two sequences \(I\) and \(J\) is written as \(IJ\), and the reverse of a sequence \(I\) is written as \(\text{REV}(I)\). If \(J\) is a sequence of requests, we use \(s \times (J)\) to denote the concatenation of \(s\) copies of list \(J\). So, for example, \(3 \times (J)\) denotes \(JJJ\). In notation referring to a particular request sequence, we will often indicate the sequence as a superscript.

For any index \(j \in \{1, \ldots, |I|\}\), let \(\text{PREV}^I(j)\) denote the index \(j' < j\) of the latest request in \(I\) such that \(I_j = I_{j'}\), if it exists. Thus, there are no requests to that item in \(I_{j'}, \ldots, I_{j-1}\). Similarly, let \(\text{SUCC}^I(j)\) denote the index of the earliest request to \(I_j\) after \(j\), if it exists.

We define the **distance**, \(d^I(j)\), of the \(j\)th request in \(I\) to be the cardinality of the set of items in requests from index \(\text{PREV}^I(j)\) to index \(j\), i.e.

\[
|\{I_{\text{PREV}^I(j)}, \ldots, I_j\}| = |\{I_{\text{PREV}^I(j)}, \ldots, I_{j-1}\}|,
\]

if \(\text{PREV}^I(j)\) exists, and \(\ell\) otherwise. Note that multiple requests to the same item in such a sequence are only counted once. Similar definitions of distance have been used for Paging [3], for example.

We extend the notation to sets and sequences of indices so that \(d^I(S) = \sum_{j \in S} d^I(j)\) is the sum of distances for all indices in a set or sequence, \(S\), of indices into a request sequence \(I\). We define \(d(I) = d^I(\{1, \ldots, |I|\})\) and refer to this as the **total** distance of \(I\).

The distance measure very closely reflects MFT’s cost on a sequence. In the upper bound proof, it is used as a tool for measuring the difference in cost incurred by MFT on two different merges.
Lemma 3  For any request sequence \( I \),
\[
d(I) - \frac{1}{2} \ell^2 + \frac{1}{2} \ell \leq \text{MTF}(I) \leq d(I).
\]

Proof  Consider a request \( I_x \) in \( I \) that is not the first request to one of the items. The cost for MTF to serve \( I_x \) is exactly \( d^I(x) \), since it has moved \( d^I(x) - 1 \) different items in front of the item requested by \( I_x \) since it last moved that item to the front of the list.

By definition, the distance of the first request to each item is \( \ell \), which is an upper bound on the cost for MTF to serve the request, since the list has length \( \ell \). Thus, the total distance \( d(I) \) is an upper bound on the cost for MTF to serve all the requests in \( I \).

Suppose that \( I \) contains requests to \( k \leq \ell \) different items. The item originally in position \( j \) in the list will be in position \( j \) or larger when it is requested the first time, so the cost of the first request to that item is at least \( \sum_{j=1}^{k} j = \frac{1}{2} (k^2 + k) \). Since \( d(I) - k\ell \) is the cost for all subsequent requests to items, \( d(I) - k\ell + \frac{1}{2}(k^2 + k) \geq d(I) - \frac{1}{2} \ell^2 + \frac{1}{2} \ell \) is a lower bound on the cost for MTF to serve all the requests in \( I \).

Next, we give a lower bound on the cost of OPT in terms of the distance measure. We use the fact that the strict competitive ratio of MTF is \( 2 - \frac{2}{\ell+1} \)[27].

Lemma 4  For any request sequence \( I \) of length \( n \),
\[
\text{OPT}(I) \geq \frac{d(I) \ell + 1}{2} - \frac{\ell^2 - 1}{4}
\]

Proof  
\[
\text{OPT}(I) \geq \frac{\text{MTF}(I)}{2 - \frac{2}{\ell+1}}, \text{by the competitiveness of MTF}
\]
\[
\geq \frac{d(I) - \frac{1}{2} \ell^2 + \frac{1}{2} \ell}{2 - \frac{2}{\ell+1}}, \text{by Lemma 3}
\]
\[
= \frac{d(I) \ell + 1}{2} - \frac{\ell^2 - 1}{4}.
\]

A merge of some request sequences is an interleaving of their requests, respecting the order of each sequence, as in mergesort. We often let \( M \) denote a merged sequence, and \( C \) the particularly simple merge which is the concatenation of the sequences in order of their process numbers. The merge will be fixed throughout proofs, so we do not add it to the notation.

For the fully adversarial scheduler model, we reduce the problem of comparing DMTF to OPT at the operation level to considering the ratio of the distances of two merges of the request sequences. In the next section, we present tight bounds on the ratio \( \frac{d(M_1)}{d(M_2)} \), for two different merges \( M_1 \) and \( M_2 \) of the same \( p \) request sequences.

The following theorem reduces the competitiveness problem to that of determining the worst-case distance ratio. We use that the strict competitive ratio of MTF is \( 2 - \frac{2}{\ell+1} \)[27].

Theorem 5  When MTF is processing the sequence \( M_1 \) and OPT is processing the sequence \( M_2 \), both of the same length, then the ratio of MTF to OPT is at most \( (2 - \frac{2}{\ell+1}) \frac{d(M_1)}{d(M_2)} \).
Proof From Lemmas 3 and 4 we get a ratio of at most
\[
\frac{d(M_1)}{d(M_2) \ell + 1} = (2 - \frac{2}{\ell + 1}) \frac{d(M_1)}{d(M_2)}.
\]

\[\square\]

In the next section, we show an asymptotically tight result, stating that the ratio, \(\frac{d(M_1)}{d(M_2)}\), is \(2p^2 - p\) in the worst case for large \(\ell\).

4 Worst-Case Distance Ratio Between Merges

In this section, we find the exact worst-case ratio, up to an additive constant, between the total distances of two different merges of the same \(p\) sequences. The results are the main technical contributions of this paper.

4.1 Lower Bound

We show that the maximal ratio of the average distances for two merges of the same \(p\) sequences of requests to a list of length \(\ell\) is approximately \(2p^2 - p\) for \(\ell\) significantly larger than \(p\).

**Theorem 6** The maximum ratio of the average distances for two merges of the same \(p\) sequences of requests to a list of length \(\ell\) is at least \(2p^2 - p - 4(p^4 - p^3)\ell + 2p^2 - p\).

**Proof** For ease of presentation, we assume that \(p\) divides \(\ell\).

For the request sequences we construct, the basic building blocks are \(A^{(j)} = \{j - 1\frac{\ell}{p} + 1, \ldots, j\frac{\ell}{p}\}\), for \(j \in \{1, \ldots, p\}\), each of which is a sequence of requests to \(\ell/p\) consecutive items. Define block \(B^{(j)}\), for \(j \in \{1, \ldots, p\}\), to be \(s\) repetitions of the concatenation of \(A^{(j)}\) and \(\text{rev}(A^{(j)})\), i.e., \(B^{(j)} = s \times (A^{(j)} \text{rev}(A^{(j)}))\).

The request sequence for process \(i\) is now defined to be
\[
\sigma_i = r \times (B^{(i)} B^{((i \text{ mod } p) + 1)} B^{((i+1 \text{ mod } p) + 1)} \ldots B^{((i+p-2 \text{ mod } p) + 1)}),
\]
which consists of \(r\) repetitions of requests to all of the \(p\) different blocks, in order, starting with block \(B^{(i)}\). Thus, each process’s sequence consists of \(rp\) blocks.

Let \(\text{Block}(\sigma_i, h)\) denote the \(h\)th block in \(\sigma_i\). Thus, \(\text{Block}(\sigma_i, h)\) is block \(B^{(j)}\) for some \(j\).

Since each process starts with a different block, it follows that, for any fixed \(h\), each of the \(p\) blocks \(\text{Block}(\sigma_1, h), \ldots, \text{Block}(\sigma_p, h)\) contains requests to a disjoint set of \(\ell/p\) items. Thus, one possible merge is
\[
prs \times (A^{(1)} \ldots A^{(p)} \text{rev}(A^{(1)}) \ldots \text{rev}(A^{(p)})
\]
where \(A^{(1)} \ldots A^{(p)}\) is \(1, \ldots, \ell\).

The average distance of this merge is close to \(\ell\). We now compute the exact value. Consider \(A^{(1)}\) and \(\text{rev}(A^{(1)})\), which have the sequence \(\frac{\ell}{p} + 1, \ldots, \ell\) in between them in the merge:
\[
1, 2, \ldots, \frac{\ell}{p}, \frac{\ell}{p} + 1, \ldots, \ell, \frac{\ell}{p}, \frac{\ell}{p} - 1, \ldots, 2, 1
\]
Considering the last \( \ell/p \) requests, the distance for the request to item 1 is \( \ell \), the distance for the request to item 2 is \( \ell - 1 \), and the distance for the request to item \( \ell p \) is \( \ell + 1 - \ell p \). Thus, the average distance over the \( \ell p \) items in \( \text{REV}(A^{(1)}) \) is 
\[
\ell - \frac{1}{\ell p} \sum_{i=1}^{\ell p} (i - 1) = \ell - \frac{\ell}{2p} + \frac{1}{2} = \frac{(2p-1)\ell + p}{2p}.
\]
Note that this holds up to renaming of the items for any pair \( (A^{(i)}, \text{REV}(A^{(i)})) \) or \( (\text{REV}(A^{(i)}), A^{(i)}) \).

We now create another merge. For \( p \leq h \leq rp \), we observe that
\[
\text{Block}(\sigma_1, h) = \text{Block}(\sigma_2, ((h - 2) \mod p) + 1) = \cdots = \text{Block}(\sigma_p, ((h - p) \mod p) + 1).
\]

We can merge each such collection of \( p \) identical blocks into
\[
s \times (p \times (e), p \times (e + 1), \ldots, p \times (e + (\frac{\ell}{p} - 1))), p \times (e + (\frac{\ell}{p} - 1)), \ldots, p \times (e)
\]
where \( e \) is the first item in the blocks. We can do this \( rp - p + 1 \) times, starting with block \( \text{Block}(\sigma_1, p) \). This will leave some blocks at the beginning and/or end of each sequence unused (specifically, the first \( p - 1 \) blocks of \( \sigma_1 \), the first \( p - 2 \) blocks and last block of \( \sigma_2 \), the first \( p - 3 \) blocks and last 2 blocks of \( \sigma_3 \), \ldots, the first block and last \( p - 2 \) blocks of \( \sigma_{p-1} \), and the last \( p - 1 \) blocks of \( \sigma_p \)).

The number of items, \( c_1 \), as well as the sum of distances, \( c_2 \), in the merge of items in the unused blocks are a function of \( \ell, p \), and \( s \), but independent of \( r \).

For each of the \( rp - p \) merges of \( p \) blocks, the first time we consider a given item of the \( \ell p \) different ones, the distance to the previous occurrence of that item may be as much as \( \ell \). In the remaining \( s - 1 \) repetitions, the distance varies between 1 and \( \ell p \) due to the reversal, with an average of \( \frac{\ell}{2p} + \frac{1}{2} \), when we consider a given item again (this average also holds for the second time one sees an item in the first repetition). Repeating an item \( p \) times gives additional total distance \( p - 1 \) for the \( p - 1 \) repetitions.

In total, the distance is at most
\[
(rp - p) \left( \frac{\ell}{p} \left( \ell + \frac{\ell}{2p} + \frac{1}{2} \right) + (s - 1) \left( \frac{2\ell}{p} \left( \frac{\ell}{2p} + \frac{1}{2} \right) \right) + s \frac{2\ell}{p} (p - 1) \right) + c_2
\]
and the number of items is
\[
(rp - p)s \frac{2\ell}{p} p + c_1 = 2(rp - p)s\ell + c_1,
\]
where \( c_2 \) does not depend on \( r \) and \( c_1 \) does not depend on either \( r \) or \( s \).

We now consider the limit of the average distances as \( r \) and \( s \) approach infinity:
\[
\lim_{s \to \infty} \lim_{r \to \infty} \frac{(rp - p) \left( \frac{\ell}{p} \left( \ell + \frac{\ell}{2p} + \frac{1}{2} \right) + (s - 1) \left( \frac{2\ell}{p} \left( \frac{\ell}{2p} + \frac{1}{2} \right) \right) + s \frac{2\ell}{p} (p - 1) \right) + c_2}{2(rp - p)s\ell + c_1}
\]
\[
= \lim_{s \to \infty} \frac{\ell \left( \ell + \frac{\ell}{2p} + \frac{1}{2} \right) + (s - 1) \left( \frac{2\ell}{p} \left( \frac{\ell}{2p} + \frac{1}{2} \right) \right) + s \frac{2\ell}{p} (p - 1)}{2s\ell}
\]
\[
= \frac{\frac{2\ell}{p} \left( \ell + \frac{1}{2} \right) + \frac{2\ell}{p} (p - 1)}{2\ell}
\]
\[
= \frac{\ell + 2p^2 - p}{2p^2}
\]
13
Considering the ratio of the average distances for the two merges, we get a value of at least

\[
\frac{(2p-1)\ell+p}{2p} = \frac{p(2p-1)\ell+p^2}{\ell+2p^2-p} = 2p^2 - p - \frac{4(p^4 - p^3)}{\ell + 2p^2 - p}.
\]

\[\qed\]

4.2 Upper Bound

In this section, we prove a matching upper bound, showing that, up to an additive constant, the maximal ratio for two merges of the same \(p\) sequences of requests to items in a list of length \(\ell\) is \(2p^2 - p\), up to an additive constant, which depends only on \(p\) and \(\ell\). To do this, we first bound how much larger the distance of the concatenation of the \(p\) sequences can be compared to the distance of any merge of these sequences.

**Lemma 7** Suppose \(C\) is the concatenation of \(p\) request sequences and \(M\) is any merge of the \(p\) sequences. Then

\[d(C) \leq p \cdot d(M)\]

**Proof** We denote the \(p\) request sequences by \(\sigma_1, \ldots, \sigma_p\).

For a request sequence \(I\) and an item \(a\), let \(item^I(a)\) denote the sum of distances of the requests to item \(a\), i.e.,

\[item^I(a) = \sum_{I_i=a} d^I(i)\]

Then, \(item^C(a) \leq \sum_{i=1}^p item^{\sigma_i}(a)\), since the only \(a\)'s changing distance in \(C\) are the first ones in each of the sequences \(\sigma_2, \ldots, \sigma_p\), where their distance was the maximal \(\ell\). Let \(max\) be the index of a maximal term in this sum, i.e., \(item^{\sigma_{max}}(a) \geq item^{\sigma_i}(a)\) for any \(i\). Then \(item^C(a) \leq p \cdot item^{\sigma_{max}}(a)\).

For any subsequences \(S\) and \(S'\), and for any \(b\), possibly equal to \(a\), \(item^{aSS'a}(a) \leq item^{aS'bS'a}(a)\).

So for any \(\sigma_i\), in particular for the maximal index \(max\) chosen above, we can merge in all items from the other sequences one at a time without reducing the sum of the \(a\)-distances, in the end obtaining \(M\). Thus, \(item^{\sigma_{max}}(a) \leq item^M(a)\).

Combining the two inequalities, \(item^C(a) \leq p \cdot item^M(a)\), and summing over all items gives the result.

The more difficult result is to bound how much larger the distance of any merge of the \(p\) sequences can be compared to the distance of their concatenation.

4.2.1 Intuitive Proof Overview

As intuition for the upper bound in the case where the merge of the \(p\) sequences gives a larger total distance than the concatenation, we consider an example which is a simplification of the construction used for the lower bound. Recall that

\[A^{(j)} = (j-1)\frac{\ell}{p} + 1, \ldots, j\frac{\ell}{p}\]

where the concatenation of \(A^{(1)}, \ldots, A^{(p)}\) is \(1, \ldots, \ell\). The request sequence of process \(j\) is \(\sigma_j = B^{(j)} = s \times (A^{(j)} \text{ Rev}(A^{(j)}))\), which consists of \(s\) repetitions of the concatenation of \(A^{(j)}\) and \(\text{ Rev}(A^{(j)})\).

The merge, \(M\), we consider is \(s \times (A^{(1)} \cdots A^{(p)} \text{ Rev}(A^{(1)}) \cdots \text{ Rev}(A^{(p)}))\).
In $M$, between consecutive copies of $A^{(j)}$ and $\text{REV}(A^{(j)})$ or between $\text{REV}(A^{(j)})$ and $A^{(j)}$ from $\sigma_j$, there are requests to $\frac{4}{p}$ distinct items from each of the $p$ other sequences. Thus, the distance in $M$ of one request coming from $\sigma_j$ to the previous request to the same item is the sum of the distance between those two requests in $\sigma_j$ plus $(p-1)\frac{4}{p}$. The asymptotic average distance of the merge, $M$, of the $p$ request sequences is $\frac{(2p-1)p+4p}{2p}$. The asymptotic average distance of the concatenation, C, of the $p$ request sequences is $\frac{2p}{2p}$. Thus, the total distance in the merge, $d(M)$, is, asymptotically, a factor of at most $2p - 1$ larger than the total distance in the concatenation, $d(C)$.

We now introduce notation used in the proof. If a number of sequences including $I$ and $J$, which, possibly together with additional sequences, are merged into $M$, and an index $h$ in $I$, we define $S^I_h$ to be

$$\{j \in \{1, \ldots, |J|\} \mid f^I(h) < f^I(j) < f^I(\text{succ}^I(h)) \text{ and } J_j \neq J_{j'} \text{ for all } j' < j \text{ where } f^I(h) < f^I(j')\},$$

the set of all indices of requests in $J$ which are merged into $M$ after the request of index $f^I(h)$, but before the request of index $f^I(\text{succ}^I(h))$, and which are first occurrences of requests to items with this property.

Recall that $d^i(\text{succ}^{\sigma_j}(x))$ is the distance from $\text{succ}^{\sigma_j}(x)$ to $x$ in $\sigma_j$. Now we can express the distance from $\text{succ}^{\sigma_j}(x)$ to $x$, in the merged sequence $M$ considered above, as $d^i(\text{succ}^{\sigma_j}(x)) + \sum_{k \neq j} |S^I_{x^j} - S^I_{x^k}|$. Summing this over all $p$ sequences $\sigma_j$ and all $x \in \sigma_j$ gives close to $(2p - 1) d(C)$.

More generally, we reduce the problem of proving the upper bound to proving that $\sum_{x \in I} |S^I_x - J_x| + \sum_{x \in J} |S^J_x - I_x| \leq 2(d(I) + d(J))$ for just two sequences, $I$ and $J$. In our example, this holds because we can match up sets of requests, since any two indexes $x$ and $x'$ in one repetition of $A^{(i)}$ have the same sets inserted between them and the next requests to the same item, so $|S^I_x - J_x| = |S^J_x - I_x|$. Thus, $\sum_{x \in I} |S^I_x - J_x| + \sum_{x' \in J} |S^J_{x'} - I_{x'}| \leq 2 \sum_{i=1}^s |A^{(i)}| \cdot |A^{(i)}|$.

In other cases, the sizes of the sets of requests merged from one sequence might not all have the same size and they might not even be merged in as blocks. However, we still partition the sequences $I$ and $J$ based (in some way) on the sets $S^I_x$ and consider the sizes of the parts of the partitions and multiply them together. If the partitions are $P^I$ and $P^J$, then we show that $\sum_{x \in I} |S^I_x - J_x| + \sum_{x' \in J} |S^J_{x'} - I_{x'}| \leq 2 \sum_{(G,G') \in (P^I,P^J)} |G| \cdot |G'|$, where the $G$ and $G'$ are corresponding pairs of parts.

It is easy to see that $2 \sum_{(G,G') \in (P^I,P^J)} |G| \cdot |G'| \leq \sum_{G \in P^I} |G|^2 + \sum_{G' \in P^J} |G'|^2$. Then, one notes, using the definitions of the partitions, that the sum of the distances for the requests in each set $G$ is at least $\frac{1}{2} \sum_{G \in P^I} |G|^2$ (and similarly for $G'$), giving the result. Note that the ordering in our example, where a set of requests is followed by requests to the same items, but in the reverse order, gives the minimum distance.

### 4.2.2 The Upper Bound Proof

First, we show that we can assume that the request sequences are disjoint.

**Lemma 8** Given sequences $\sigma_1, \ldots, \sigma_p$ referring to $\ell$ items and any merge $M$, there exist sequences $\sigma'_1, \ldots, \sigma'_p$ and a merge $M'$ such that for each $i$, $d(\sigma_i) = d(\sigma'_i)$, $d(M') \geq d(M)$, and, for any $j \neq i$, $\sigma'_i$ and $\sigma'_j$ refer to disjoint sets of a total of at most $p\ell$ items.
Proof. Below, we argue that given two lists, we can modify one of them while retaining its total distance in such a way that the number of items common to the two sequences is decreased by one and the total distance of the new, merged sequence $d(M')$ is at least $d(M)$. Using this result inductively, we can make all the sequences disjoint and have $d(M') \geq d(M)$.

Consider two lists $I = I_1, \ldots, I_{n_I}$ and $J = J_1, \ldots, J_{n_J}$ such that there is some item $a$ that occurs in both.

Let $J' = J'_1, \ldots, J'_{n_J}$ be the second list in which all occurrences of $a$ are renamed $a'$, where $a'$ does not occur in either list. Note that $d(J) = d(J')$.

Let $K' = K'_1, \ldots, K'_{n_I+n_J}$ be a merge of the lists $I$ and $J'$.

Let $K = K_1, \ldots, K_{n_I+n_J}$ be the list obtained from $K'$ by replacing all occurrences of $a'$ by $a$. Note that any merge of $I$ and $J$ can be obtained in this manner.

If $K_i$ and $K_k$ are consecutive occurrences of some item $b \neq a$, then $K'_i$ and $K'_k$ are also consecutive occurrences of $b$. Recall that $d^K(K'_i) = \left| \left\{ K'_{i,1}, \ldots, K'_{k-1} \right\} \right|$ and $d^K(K_k) = \left| \left\{ K, \ldots, K_{k-1} \right\} \right|$. If $\left\{ K'_{i,1}, \ldots, K'_{k-1} \right\}$ does not contain occurrences of both $a$ and $a'$, then $d^K(K'_i) = d^K(K_k)$; otherwise $d^K(K'_i) = 1 + d^K(K_k)$.

Now consider the locations

$$i_{1,1} < \cdots < i_{1,q_1} < i_{2,1} < \cdots < i_{2,q_2} < \cdots < i_{r,1} < \cdots < i_{r,q_r}$$

of all occurrence of $a$ in $K_{1,1}, \ldots, K_{n_I+n_J}$, where $K'_{i,j,1} = \cdots = K'_{i,j,q_j}$ for $1 \leq j \leq r$ and $K'_{i,j,1} \neq K'_{i,j+1,1}$ for $1 \leq j < r$. Then, for all $1 \leq j < r$ and $1 < k < q_j$, $d^K(K'_{i,j,k}) = d^K(K_{i,j,k})$.

For $2 < j \leq r$,

$$d^K(K_{i,j,1}) = \left| \left\{ K_{i,j-1,q_{j-1}}, K_{i,j-1,q_{j-1}+1} \ldots, K_{i,j-1,1} \right\} \right|$$

whereas

$$d^K(K'_{i,j,1}) = 1 + \left| \left\{ K'_{i,j-1,q_{j-1}}, K_{i,j-1,q_{j-1}+1} \ldots, K_{i,j-1,1} \right\} \right|$$

Note that $d^K(K_{i,1}) = 1 + |\left\{ K_h \mid i_{1,q_1} < h < i_{2,1} \right\}|$, but $d^K(K'_{i,1}) = \ell$, since it is the first occurrence of this item.

Thus,

$$d(K') \geq d(K) + \ell - d^K(K_{i,1}) \geq d(K)$$

Introducing the new item $a'$ in the above increases the total number of items by one. Doing so $\ell$ times for $p-1$ sequences increases the number of items to at most $p\ell$. \hfill \Box

For the proof sequence to follow, we first consider two request sequences and then later reduce the proof for $p$ sequences to the result for two sequences.

We use the following algorithms, Algorithms 1 and 2, to define partitions $P_I$ and $P_J$ of the indices of two request sequences, $I$ and $J$, such that all indices in a partition represent requests to distinct items, i.e., if indices $i$ and $j$ are in the same part of a partition of $I$, then $I_i \neq I_j$. We slightly abuse
the term partition since not all requests in \( J \) are necessarily included in a part in \( P^J \), but \( P^J \) is a partition of a subsequence of \( J \). In the algorithms, we refer to an index as \emph{unassigned} as long as it has not been assigned to any part in a partition.

**Algorithm 1** Creating the partition for \( I \).

1: \( P^I \leftarrow \emptyset \)
2: \( \text{cnt} \leftarrow 0 \)
3: \( \textbf{while} \) there are unassigned indices in \( I \) \( \textbf{do} \)
4: \( i \leftarrow \text{first such index} \)
5: \( \text{cnt} \leftarrow \text{cnt} + 1 \)
6: \( G^I_{\text{cnt}} \leftarrow \{i\} \)
7: \( \textbf{if} \) there exists an index \( j > i \) such that \( I_i = I_j \) \( \textbf{then} \)
8: \( j \leftarrow \text{smallest such index} \)
9: \( \textbf{for} \ h \in i + 1, \ldots, j - 1 \ \textbf{do} \)
10: \( \textbf{if} \ h \text{ is unassigned and } I_h \notin \{I_{i+1}, \ldots, I_{h-1}\} \textbf{then} \)
11: \( G^I_{\text{cnt}} \leftarrow G^I_{\text{cnt}} \cup \{h\} \)
12: \( P^I \leftarrow P^I \cup \{G^I_{\text{cnt}}\} \)

**Algorithm 2** Creating the partition for \( J \) based on \( I \).

1: \( P^J \leftarrow \emptyset \)
2: \( q \leftarrow \text{number of parts in } P^I \)
3: \( \textbf{for} \ \text{cnt} \text{ in } 1, \ldots, q \ \textbf{do} \)
4: \( G^J_{\text{cnt}} \leftarrow \emptyset \)
5: \( i_1, \ldots, i_q \leftarrow \text{ordered sequence of indices in } G^I_{\text{cnt}} \)
6: \( \textbf{for} \ j \text{ in } 1, \ldots, q \ \textbf{do} \)
7: \( G^J_{\text{cnt}} \leftarrow G^J_{\text{cnt}} \cup \{h \in S^I_{i_j} \mid h \text{ is unassigned and } J_h \notin \bigcup_{k \in \{1, \ldots, j-1\}} \{J_l \mid l \in S^I_{i_k}\}\} \)
8: \( P^J \leftarrow P^J \cup \{G^J_{\text{cnt}}\} \)

Thus, with reference to two fixed request sequences \( I \) and \( J \), a merge of them, \( M \), and the partitions defined by the algorithms, the parts of \( P^I \) are enumerated in order of creation as \( G^I_1, G^I_2, \ldots, G^I_q \). The creation of each part in \( P^J \) is triggered by a part of \( P^I \). For these parts, \( G^I_1, G^I_2, \ldots, G^I_q \), the \( G^J_j \) triggered the creation of \( G^I_j \), and we refer to these as \emph{corresponding} parts.

We let \( \mathcal{P} \) denote the set of all pairs of indices in the Cartesian product of corresponding parts. Thus, if there are \( q \) parts in \( P^I \),
\[
\mathcal{P} = \{(i, j) \mid h \in \{1, \ldots, q\}, i \in G^I_h, j \in G^J_i\}.
\]

**Lemma 9** Defining sets and partitions, following Algorithms 1 and 2 from two disjoint request sequences \( I \) and \( J \), which are merged into \( M \), there exists an injective mapping from \( \{(x, y) \mid x \in \{1, \ldots, |I|\}, y \in S^I_{x-1} \} \) to \( \mathcal{P} \).

**Proof** Assume that \( y \in S^I_{x-1} \). Since \( P^I \) is a partition, there exists a unique part \( G^I \in P^I \) such that \( x \in G^I \). Let \( G^J \in P^J \) be the corresponding part, i.e., the part, the creation of which was triggered by \( G^I \) in Algorithm 2.

\underline{Case } \( y \in G^I \): If \( y \in G^J \), we map \((x, y)\) to \((x, y)\). For the remainder of the proof, we assume that \( y \notin G^J \).
Case $y \notin G^J$ and $y$ is not in any part of $P^J$:  
Since $y \in S^{I_{x,J}-I}_x$, $y$ is the index from $J$ to the first occurrence of an item $J_y$ appearing after $f^I(x)$ in $M$. We first observe that the index $x$ cannot be the first added to its part in Algorithm 1. If it were, then $y$ would be added to a part in Algorithm 2, since $y$ is unassigned (by assumption) and the rightmost argument is trivially true the first time through the for-loop (Line 7 of Algorithm 2), so $y$ would be in $G^J$. Thus, assume an earlier index $z$ into $I$ was the first to be placed in the part where $x$ was added later.

Since $x$ is added to $z$’s part, there are no other requests to the item $I_x$ between $z$ and $x$ in $I$.

Since $y$ is not in any part, that is, unassigned, there must be another request to $J_y$ between $f^I(z)$ and $f^I(x)$ in $M$, preventing $y$ from being added. Let $y'$ be the first index into $J$ such that $f^I(y')$ comes after $f^I(z)$ in $M$ and $J_y = J_{y'}$. By definition, since $I$ and $J$ are disjoint, $y' \in G^J$. We map $(x, y)$ to $(x, y')$, and since $y' \notin S^{I_{x,J}-I}_x$, no other pair can be assigned to $(x, y') \in \mathbb{P}$.

Case $y \notin G^J$, but $y$ is in some part:
Let $z$ be the request in $I$ which starts the part $H^I$ in $P^I$ which gives rise to the part $H^J$ in $P^J$ to which $y$ belongs. There must be a request to item $I_x$ between $z$ and $x$, since otherwise, $x$ would be in the same part as $z$ and $y$ would then belong to $G^J$. Let $x'$ be the first index after $z$ such that $I_x = I_{x'}$. We assign $(x, y)$ to $(x', y)$. Since $y$ belongs to the part of $P^J$, the creation of which was triggered by the part containing $z$, there cannot be any request to any item equal to $J_y$ between $f^I(z)$ and $f^I(y')$ in $M$, so no other pair can be assigned to $(x', y)$. Since $x' \in H^I$, $x'$ and $y$ are again in corresponding parts. □

Lemma 10 For two disjoint request sequences $I$ and $J$ together with their merge $M$,

$$\left| \{(x, y) \mid y \in S^{I_{x,J}-I}_x \} \right| \geq \left| \{(x, y) \mid x \in S^{J_{y,I}-I}_y \} \right| - \ell^2.$$

Proof Consider a pair $(x, y)$ such that $x \in S^{J_{y,I}-I}_y$. Assume that there are requests to the item $I_y$ after $y$ in $J$. Let $y'$ be the first index larger than $y$ in $J$ such that $I_{y'} = I_y$ and let $x'$ be the last index in $I$ such that $I_x = I_{x'}$ and $f^I(x')$ comes before $f^I(y')$ in $M$ ($x'$ could be $x$). Clearly, $y' \in S^{I_{x,J}-I}_x$, so $(x', y') \in \{(x, y) \mid y \in S^{I_{x,J}-I}_x \}$. The mapping $(x, y) \mapsto (x', y')$ is injective since no index $x''$ between $x$ and $x'$ such that $I_{x''} = I_x$ can belong to $S^{I_{x,J}-I}_y$. The mapping is defined for all but at most $\ell$ requests in $J$ (the last request to each item) and at most $\ell$ $x$-values for each such request $y$ in $J$, which amounts to at most $\ell^2$ pairs. □

Lemma 11 For two disjoint request sequences $I$ and $J$ together with their merge $M$ and partitions $P^I$ and $P^J$,

$$\sum_{1 \leq x \leq |I|} |S^{I_{x,J}-I}_x| + \sum_{1 \leq y \leq |J|} |S^{J_{y,I}-I}_y| \leq 2 |\mathbb{P}| + \ell^2.$$

Proof

$$\sum_{1 \leq x \leq |I|} |S^{I_{x,J}-I}_x| = \left| \{(x, y) \mid x \in \{1, \ldots, |I|\}, y \in S^{I_{x,J}-I}_x \} \right|,$$

which, by Lemma 9, is at most $|\mathbb{P}|$.

By Lemma 10 $\left| \{(x, y) \mid y \in S^{I_{x,J}-I}_x \} \right| \geq \left| \{(x, y) \mid x \in S^{J_{y,I}-I}_y \} \right| - \ell^2$. So, we get that $\sum_{1 \leq y \leq |J|} |S^{J_{y,I}-I}_y| \leq |\mathbb{P}| + \ell^2$.

Adding the two bounds gives the result. □

We need the following simple lemma:

Lemma 12 If $I$ is a sequence of requests to distinct items and $J$ can be any permutation of $I$, then $d^I(J)$ is minimized when $J = \text{REV}(I)$.  

18
**Proof** Suppose to the contrary that the minimum is attained for some \( J \neq \text{REV}(I) \).

Let \( J = (x_1, x_2, \ldots, x_m) \). Consider the largest value \( k \) such that \( x_k \) and \( x_{k+1} \) are indices of items occurring in the same order in \( I \). Define

\[
J' = (x_1, \ldots, x_{k-1}, x_{k+1}, x_k, x_{k+2}, \ldots, x_m),
\]

where only these two items are swapped. Then,

\[
d''^J(I) = d^J(I) + (d''^J(x_{k+1}) - d''^J(x_k)) + (d''^J(x_k) - d^J(x_k))
\]

\[
= d^J(I) - 1
\]

This follows since items originating from the \( x_k \)'s no longer have the \( x_{k+1} \)-items between them, and the two copies of \( x_k \)-items between the \( x_{k+1} \)-items are only counted once. Thus, \( J \) did not give rise to the minimum value, giving the contradiction.

**Lemma 13** For two request sequences \( I \) and \( J \) together with their merge \( M \) and partitions \( P^I \) and \( P^J \), 

\[
\sum_i |G_i^I| \cdot |G_i^J| \leq \sum_i d^I(G_i^I) + \sum_i d^J(G_i^J) + 3f^2.
\]

**Proof** By simple arithmetic, for any two positive values, \( a \) and \( b = a - c \) for some \( c \), \( 2ab = 2a^2 - 2ac \leq 2a^2 - 2ac + b^2 = a^2 + b^2 \). By this, we conclude that

\[
2 \sum_i |G_i^I| \cdot |G_i^J| \leq \sum_i |G_i^I|^2 + \sum_i |G_i^J|^2.
\]

We can relate \( |G_i^I|^2 \) to \( d^I(G_i^I) \), and similarly for \( J \). We prove the following properties where \( K \) can be either \( I \) or \( J \):

- The items requested in \( G_i^K \) are all distinct.
- If \( q \) is the first index added to \( G_i^K \), then the other items requested in \( G_i^K \) are each the first occurrence of that item after \( q \) in \( K \).

For \( G_i^I \), the index \( h \) is only added if \( I_h \notin \{I_{q+1}, \ldots, I_{h-1}\} \), where \( h \) is the next index for a request to \( J_q \), and the indices \( q + 1, \ldots, h - 1 \) are the only smaller indices considered for \( G_i^I \). Thus, both properties hold for \( K = I \).

For \( G_i^J \), the index \( h \) is only added if

\[
J_h \notin \bigcup_{k \in \{1, \ldots, j-1\}} \{J_l \mid l \in S_{i_k}^{l_{i-J}}\},
\]

where the items indexed in each \( S_{i_k}^{l_{i-J}} \) are distinct and \( \bigcup_{k \in \{1, \ldots, j-1\}} S_{i_k}^{l_{i-J}} \) are the only smaller indices considered for \( G_i^J \), other than smaller indices in \( S_{i_j}^{l_{i-J}} \). Thus, the first property holds.

For the second property, assume for the sake of contradiction that \( q \) is the smallest index added to \( G_i^J \), \( q < j_1 < j_2 \), \( J_{j_1} = J_{j_2} \), with \( j_1 \) being the smallest index having this property, and \( j_2 \in G_i^J \). Let \( x \) be the smallest index in \( G_i^J \). Now, \( j_1 \) is not in \( S_x^{l_{i-J}} \), since then \( j_2 \) could not belong to \( G_i^J \), by Algorithm 2. Hence, there exists an index \( x' \) with \( I_x = I_{x'} \) after \( x \), which is merged into \( M \) before \( j_1 \). For \( j_2 \) to belong to \( G_i^J \), there must exist a \( y \) in \( I \) between \( x \) and \( x' \) such that \( j_2 \) is in \( S_y^{l_{i-J}} \). However, since \( y \) appears before \( x' \), \( j_1 \) would be in \( S_y^{l_{i-J}} \), preventing \( j_2 \) from belonging to \( S_y^{l_{i-J}} \), arriving at a contradiction.
The following argument holds for any of the parts from either $I$ or $J$ and we use $K$ to denote either.

Consider the sequence $X^K_i$ of requests in $G^K_i$, for which there is a previous request to the same item in $K$, and let the requests in $X^K_i$ occur in the order they occur in $K$. Further, let $x$ be the smallest index among them. The requests in $G^K_i$, and therefore also in $X^K_i$, are all to distinct items, and they are each the first request to that item after index $x$. This means that for any request of $X^K_i$, the previous request to it, if any, occurred before the request at index $x$.

The total distance between the requests indexed by $X^K_i$ and their previous occurrences is at least \[ a = \min_Y d^Y X^K_i (X^K_i), \] where $Y$ is any permutation of $X^K_i$. By Lemma 12, the minimum value, $a$, is obtained when $Y$ is REV($X^K_i$).

For the special permutation $\text{REV}(X)$, $d^{\text{REV}(X)}(X) = |X| \cdot (|X| + 1)/2$. Thus, for any $X^K_i$, one can assume that $|X^K_i|^2 \leq 2 d^K(X^K_i)$.

Now,
\[
\begin{align*}
\sum_i |G^K_i|^2 + \sum_i |G^K_i|^2 &= \sum_i |X^K_i|^2 + \sum_i |G^K_i \setminus X^K_i|^2 + 2 \sum_i |G^K_i| \cdot |G^K_i \setminus X^K_i| + \\
&\leq \sum_i |X^K_i|^2 + 3 \sum_i |G^K_i| \cdot |G^K_i \setminus X^K_i| + \\
&\leq 2 \sum_i d^l(X^K_i) + \sum_i 2 d^l(X^K_i) + 6 \ell^2 \\
&\leq 2 \sum_i d^l(G^K_i) + 2 \sum_i d^l(G^K_i) + 6 \ell^2.
\end{align*}
\]

In the first equality, we abuse notation slightly and treat sequences as sets in the obvious way. The second inequality follows since, for each part, there are at most $\ell$ requests in total from all $G^K_i$ together which are not counted in any $X^K_i$ because they did not have previous requests, and since each $G^K_i$ has size at most $\ell$.

Combining this with Eq. 1
\[
2 \sum_i |G^K_i| \cdot |G^K_i| \leq 2 \sum_i d^l(G^K_i) + 2 \sum_i d^l(G^K_i) + 6 \ell^2,
\]
and the lemma follows. □

**Lemma 14** For any integer $p \geq 2$, there exists a constant $c$, depending on $p$ and $\ell$, such that for all disjoint request sequences $\sigma_1, \sigma_2, \ldots, \sigma_p$,
\[
d(M) \leq (2p - 1) d(C) + c,
\]
where $C$ is the concatenation of $\sigma_1, \sigma_2, \ldots, \sigma_p$ and $M$ is any merge of these $p$ sequences.

**Proof** For the concatenation, $C$, of the $p$ sequences, the sum of all the distances in $C$ is $d(C) \leq \sum_{i=1}^p d(\sigma_i)$, because the sequences are disjoint.

Recall that the distance of some index $h$ in $\sigma_j$, $d^\sigma_j(h)$, is the number of distinct requests between $h$ and PREV$\sigma_j(h)$ if PREV$\sigma_j(h)$ exists and $\ell$ otherwise.
Due to disjointness, this distance in $\sigma_j$ is only increased in $M$ by the requests from other sequences that are inserted between those two requests. Thus, the distance between $f^{\sigma_j}(h)$ and $f^{\sigma_j}(\text{PREV}^{\sigma_j}(h))$, is at most $d^{\ell_j}(h) + \sum_{k \neq j} |S^{\sigma_j \leftarrow \sigma_k}_{\text{PREV}^{\sigma_j}(h)}|$. If $\text{PREV}^{\sigma_j}(h)$ does not exist, we define $|S^{\sigma_j \leftarrow \sigma_k}_{\text{PREV}^{\sigma_j}(h)}|$ to be zero.

Using this,

$$d(M) = \sum_{i \in \{1, \ldots, |M|\}} d^M(i)$$
$$= \sum_{j=1}^p \sum_{h \in \{1, \ldots, |\sigma_j|\}} d^M(f^{\sigma_j}(h))$$
$$\leq \sum_{j=1}^p \sum_{h \in \{1, \ldots, |\sigma_j|\}} (d^{\ell_j}(h) + \sum_{k \neq j}^p |S^{\sigma_j \leftarrow \sigma_k}_{\text{PREV}^{\sigma_j}(h)}|)$$
$$= \sum_{j=1}^p (d(\sigma_j) + \sum_{h \in \{1, \ldots, |\sigma_j|\}} \sum_{k \neq j}^p |S^{\sigma_j \leftarrow \sigma_k}_{\text{PREV}^{\sigma_j}(h)}|)$$

By the interpretation of $|S^{\sigma_j \leftarrow \sigma_k}_{\text{PREV}^{\sigma_j}(h)}|$ when $\text{PREV}^{\sigma_j}(h)$ does not exist,

$$\sum_{h \in \{1, \ldots, |\sigma_j|\}} \sum_{k=1}^p |S^{\sigma_j \leftarrow \sigma_k}_{\text{PREV}^{\sigma_j}(h)}| \leq \sum_{h \in \{1, \ldots, |\sigma_i|\}} \sum_{k=1}^p |S^{\sigma_i \leftarrow \sigma_k}_{h}| .$$

Since $\sum_{i=1}^p d(\sigma_i) \leq d(C)$, as stated as the first remark of the proof, we are done if we can establish that

$$\sum_{i=1}^p (d(\sigma_i) + \sum_{h \in \{1, \ldots, |\sigma_i|\}} \sum_{k \neq i}^p |S^{\sigma_i \leftarrow \sigma_k}_{h}|) \leq (2p - 1) \sum_{i=1}^p d(\sigma_i) + 7p^2 \ell^2,$$

where $7p^2 \ell^2$ is the constant $c$ in the lemma statement. This is equivalent to showing that

$$\sum_{h \in \{1, \ldots, |\sigma_i|\}} \sum_{k \neq i}^p |S^{\sigma_i \leftarrow \sigma_k}_{h}| \leq 2(p - 1) \sum_{i=1}^p d(\sigma_i) + 7p^2 \ell^2.$$

For each sequence $\sigma_j$, we consider all $p - 1$ sequences $\sigma_k$ for which the cardinality of the sets $S^{\sigma_j \leftarrow \sigma_k}_{h}$ and $S^{\sigma_k \leftarrow \sigma_j}_{h'}$ are added into the sum for some $h$ and $h'$.

From the definition of $\mathcal{P}$, we have that $|\mathcal{P}| = \sum_i |G^i_1| \cdot |G^i_2|$. Using that in combination with Lemmas 11 and 13 gives that for all pairs, $j$ and $k$,

$$\sum_{h \in \{1, \ldots, |\sigma_j|\}} |S^{\sigma_j \leftarrow \sigma_k}_{h}| + \sum_{h' \in \{1, \ldots, |\sigma_k|\}} |S^{\sigma_k \leftarrow \sigma_j}_{h'}| \leq 2(d(\sigma_j) + d(\sigma_k)) + 7\ell^2.$$

If we sum this over all such pairs $j$ and $k$, each $j$ will appear with $p - 1$ different indices $k$, giving

$$\sum_{h \in \{1, \ldots, |\sigma_i|\}} \sum_{k \neq j}^p |S^{\sigma_k \leftarrow \sigma_j}_{h}| \leq 2(p - 1) \sum_{i=1}^p d(\sigma_i) + 7p^2 \ell^2$$

$\square$
Theorem 15. For any two merges, $M_1$ and $M_2$ of $p$ sequences of requests to a list of length $\ell$, the total distance of $M_1$ is at most $2p^2 - p$ times the total distance of $M_2$ up to an additive constant only depending on $p$ and $\ell$.

Proof. By Lemma 8, we can assume disjointness at a loss of an additive constant only depending on $p$ and $\ell$.

Let $C$ be the concatenation of the disjoint sequences $\sigma_1, \ldots, \sigma_p$.

By Lemma 14, $d(C) \leq p \cdot d(M_2)$.

By Lemma 14, $d(M_1) \leq (2p - 1) d(C) + c$, where $c$ is a constant only depending on $p$ and $\ell$.

Combining these two inequalities gives the result.

\[ \square \]

5 Interleaving at the Item Access Level – Fully Adversarial

In the previous section, we proved a bound on DMTF’s cost relative to OPT when operations are interleaved at the operation level, i.e., after the merging of the $p$ request sequences, only the cost of the sequential MTF was counted.

Now, we consider a more fine-grained interleaving; more than one process might be searching for the same item at the same time. Potentially, moving from the operation level to the item level could lead to wasted work. For instance, if two processes are searching for the same item $a$, currently in position $k$, then the cost (the number of items inspected) would be $k + 1$ when interleaving at the operation level, since one process would carry out its operation first, moving $a$ to the front before the other process would carry out its operation. The more fine-grained interleaving allows both processes to inspect the first item, then the second item, etc. Thus, the total number of items inspected could be at least $2k$.

We assume that these processes will find the item in the same location and have the same cost, rather than all but the first having cost 1, as in the operation level. We refer to this level as the item access level. We emphasize that we are not yet considering concurrency where interleaving will be even more fine-grained.

In this section, we prove that the bound derived in the previous section also holds when considering the more fine-grained interleaving. Intuitively, this is because extra work of the type described above in the fine-grained interleaving occur in situations that are not worst-case.

Theorem 16. Considering item level interleaving, for any two merges, $M_1$ and $M_2$ of $p$ sequences of requests

and any scheduling of steps to DMTF and OPT,

\[ \text{DMTF}(M_1) \leq (4p^2 - 2p) \text{OPT}(M_2) + O(1), \]

where the constant depends only on $p$ and $\ell$.

Proof. Assume that some process $p_i$ treats a request to $a$, finds it, and moves it to the front. Assume that some other process $p_j$ initiates a search for $a$ before $a$ is moved to the front and stops its search after $a$ has been moved to the front. We consider a linearization (as defined in Section 1.5) based on the order of moves to the front or discovering that an item has already been
moved to the front. Assume that in this linearization, the a of process $p_j$ is preceded by a request $b \neq a$. The linearization where we swap $a$ and $b$ can only have higher cost, since $p_j$’s search for $a$ could now pass through item $b$, since it is not moved to the front until after $p_j$ stops its search. Using this argument inductively, we can consider a worst-case linearization where where all these requests to $a$ appear in succession. We let $M$ refer to this sequence.

Our upper bound from Theorem 15 in the previous section also holds when items are renamed so that the items requested in each process are disjoint. Consider the sequence $M'$, where $a$’s from process $p_i$, $i \in \{1, \ldots, p\}$ are renamed $a_i$, where $a_1, \ldots, a_p$ is a fresh set of names.

Since $p_i$ moved $a$ to the front, $a_i$ appears first among the requests to $a_1, \ldots, a_p$ under discussion, say at index $y$ in $M'$. Let $a_k$ be the request prior to $a_i$ which appears latest in $M'$ at index $x < y$. Thus, $d^{M'}(y) \geq |\{M_x, \ldots, M_y\}|$ and the same lower bound holds for the other requests to $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_p$.

In $M$, $|\{M_x, \ldots, M_y\}|$ is the exact cost of processing the request to $a$ at index $x$. For item access level interleaving, for the other requests to $a$, we also count cost $|\{M_x, \ldots, M_y\}|$. Thus, the extra cost is already accounted for in the distances used in the proof.

By Theorem 5, the ratio at the item access level is bounded by at most twice the ratio $2p^2 - p$ from Theorem 15.

\section{DMTF: A Distributed Implementation of Move-to-Front}

This section presents DMTF, a lock-free implementation of a static set based on the well-known MTF algorithm. At the end of the description of the algorithm, a simple modification to make it wait-free is described. A complete proof of correctness is presented in in Section 6.1.

When a process finds a node containing the item it is looking for and that node is not at the front of the list, it prepends a copy of the node to the list and then removes the node it found from the list. This ensures that the list always contains a node containing each item in the set.

There are difficulties with implementing the MTF algorithm in this straightforward fashion in a distributed setting. For example, suppose that some process $p_i$ has begun looking for an item and has proceeded a few nodes along the list. If another process, which is concurrently looking for the same item, finds a node containing that item later in the list and moves it to the front of the list, then $p_i$ might reach the end of the list without finding the item. To prevent this problem, each process announces the item it is currently looking for. After a process finds the item, $e$, it is looking for and prepends a new node containing $e$ to the list, it informs all other processes that are looking for $e$ about this new node before removing the node in which it found $e$ from the list.

The processes also help one another to prepend nodes to the list and remove nodes from the list. This ensures that more than one copy of a node is not prepended to the list, for example, when multiple processes concurrently find the same node or when a process has fallen asleep for a long time and then wakes up. Additional fields in each node are used to facilitate helping.

A detailed description of our implementation is given below. Pseudocode is presented in Figures 1 and 2. Throughout the code, if $h$ is a pointer to a node and $f$ is the name of the field, then $h.f$ is a reference to that field of the node. In the description and proof of correctness, we distinguish between nodes and pointers to nodes.
function SEARCH(e) by process $p_i$

Allocate and initialize a new node.

1. $g \leftarrow \text{allocate}(\text{node})$
2. $g$.item $\leftarrow e$
3. $g$.prev $\leftarrow g$.next $\leftarrow g$.old $\leftarrow g$.new $\leftarrow \text{null}$

Announce the operation.

4. $\text{CAS}(A[i], (\text{null, } \bot), (g, e))$
5. $(h_1, h) \leftarrow \text{Head}$
6. if $h_1$.item $= e$ then
   ▷ The first node in the list contains item $e$. Moving it to the front is not necessary.
   
   7. $(a, b) \leftarrow \text{CAS}(A[i], (g, e), (\text{null, } \bot))$
   8. if $b = \bot$ then $\text{CAS}(A[i], (a, b), (\text{null, } \bot))$
   9. return $(h_1)$

Continue searching from the second node in the list.

c $\leftarrow 0$

while $h \neq \text{null}$ do

   if $h$.item $= e$ then
      ▷ A node containing item $e$ is found, but not at the front of the list.
      
      10. $(a, b) \leftarrow A[i]$
      11. if $b = \bot$ then $\text{CAS}(A[i], (a, b), (\text{null, } \bot))$
      12. return $(a)$
      
      Try to set the nodes pointed to by $g$ and $h$ to refer to one another.
      
      13. $g' \leftarrow h$.new
      14. if $g' \neq \text{gone}$ then $\text{MOVE-TO-FRONT}(g')$
      15. $(a, b) \leftarrow A[i]$
      16. if $b = \bot$ then $\text{CAS}(A[i], (a, b), (\text{null, } \bot))$
      17. return $(a)$
      
      Move the new copy of the node containing item $e$ to the front of the list.
      
      18. $h \leftarrow h$.next
      19. $c \leftarrow (c + 1) \text{ mod } \phi$
      20. if $c = 0$ then
         ▷ Some other process has changed the announcement.
         
         21. $(a, b) \leftarrow A[i]$
         22. if $b = \bot$ then $\text{CAS}(A[i], (a, b), (\text{null, } \bot))$
         23. return $(a)$

      End while
      
      ▷ End of the list was reached.
      
      24. $(a, b) \leftarrow A[i]$
      25. if $b = \bot$ then $\text{CAS}(A[i], (a, b), (\text{null, } \bot))$
      26. $h \leftarrow h$.next
      27. if $h_1$.item $= e$ then $\text{return}(a)$ else return(\text{not present})

Figure 1: An Algorithm to Search for a Node Containing the Element $e \neq \bot$
procedure MOVE-TO-FRONT($g'$) by process $p_i$
35 while $g'.old \neq DONE$ do
36 $(h_1, h_2) \leftarrow \text{Head}$
37 $h' \leftarrow h_1.old$
38 if $h' = DONE$ then
39 ▷ Try to prepend the node pointed to by $g'$ to the beginning of the list
40  if $g'.old \neq DONE$ then CAS(Head, $(h_1, h_2)$, $(g', h_1)$)
41  else  ▷ Ensure that the first two nodes in the list point to one another.
42  CAS($h_1.next$, NULL, $h_2$)
43  CAS($h_2.prev$, NULL, $h_1$)
44  $e' \leftarrow h_1.item$
45  ▷ Inform all processes looking for item $e'$
46  for every process index $j$ do
47     $(a, b) \leftarrow A[j]$
48     if $b = e'$ and $h_1.old \neq DONE$ then CAS($A[j]$, $(a, b)$, $(h_1, \perp)$)
49  ▷ Remove the node pointed to by $h'$ from the list.
50     pred $\leftarrow h'.prev$
51     succ $\leftarrow h'.next$
52     CAS(pred.next, $h'$, succ)
53     if succ $\neq$ NULL then CAS(succ.prev, $h'$, pred)
54     CAS($h'.new$, $h_1$, GONE)
55     CAS($h_1.old$, $h'$, DONE)
56 end while
57 return

Figure 2: An Algorithm for Moving a Node to the Front of the List

If the set contains at most one item, an implementation is straightforward. So, we assume that the set contains at least two items. The items of the set are stored in a doubly linked list of nodes. A compare-and-swap object, Head, contains pointers to the first and second nodes in the list. Every node in the list (excluding the first node in some intermediate configurations) contains a different item.

Each node has five fields:
- **item**, a register which contains an item and is never changed,
- **next**, a compare-and-swap object which points to the next node in the list or NULL, if the node is the last node in the list,
- **prev**, a compare-and-swap object which points to the previous node in the list or NULL, if the node is the first node in the list,
- **old**, a register which is initialized to NULL when the node is newly created, is only changed (by the process that created the node) from NULL to point to another node containing the same item, and, if it is pointing to a node, is only changed to DONE,
- **new**, a compare-and-swap object which is initialized to NULL when the node is newly created, is only changed from NULL to point to another node containing the same item, and, if it is pointing to a node, is only changed to GONE.

We assume that, initially, the old field of every node in the list is DONE, the new field of every node in the list is NULL, and the list consists of exactly one node for each item in the set, which contains that item.

Each process $p_i$ has a compare-and-swap object $A[i]$, which contains a pair. Initially, $A[i] =$
(null, ⊥), which indicates that \( p_i \) is not currently searching for an item. If the second component of \( A[i] \) is \( e \neq ⊥ \), then \( p_i \) is searching for item \( e \). In this case, the first component of \( A[i] \) is a pointer to a replacement node newly allocated by \( p_i \) at the beginning of its search and which acts as a unique identifier for the search. When the second component of \( A[i] \) is \( ⊥ \), but its first component is not \( \text{null} \), the first component is a pointer to a node that was at the front of the list at some point after \( p_i \) started its current search and contains the item \( p_i \) is searching for. Process \( p_i \) is the only process that changes the first component of \( A[i] \) from \( \text{null} \) or to \( \text{null} \) and the only process that successfully changes \( A[i] \) when its second component is \( ⊥ \).

\( \text{SEARCH}(e) \) is used by process \( p_i \) to search for a node containing the item \( e \neq ⊥ \). It begins by setting \( g \) to point to a newly allocated replacement node on line 0, setting its \( \text{item} \) field to \( e \) on line 0, and setting all its other fields to \( \text{null} \) on line 0. Next, \( p_i \) announces \((g, e)\) in \( A[i] \) on line 0 and reads \( \text{Head} \) on line 0 to get pointers to the first and second nodes in the list. Then, it goes through the list, one node at a time, in order, comparing its \( \text{item} \) with \( e \).

Suppose that, on line 0, process \( p_i \) finds that the first node in the list (i.e. the node pointed to by the first pointer in \( \text{Head} \)) contains item \( e \). Then \( p_i \) tries to reset \( A[i] \) to \((\text{null}, ⊥)\) on line 0 and returns a pointer to this node on line 0. Note that, if another process changed \( A[i] \) between when \( p_i \) announced that it was searching for \( e \) and when \( p_i \) tries to reset \( A[i] \) to \((\text{null}, ⊥)\), then the second component of \( A[i] \) will be \( ⊥ \) instead of \( e \). In this case, \( p_i \) performs a second CAS on line 0 to reset \( A[i] \) to \((\text{null}, ⊥)\).

While searching the rest of the linked list for item \( e \), process \( p_i \) repeatedly checks on line 0 whether the second component of \( A[i] \) has been set to \( ⊥ \), indicating that some other process has prepended a node containing \( e \) to the list. If so, the first component of \( A[i] \) is a pointer to such a node, which \( p_i \) returns on line 0 after resetting \( A[i] \) to \((\text{null}, ⊥)\) on line 0. It does this check each time it has examined \( φ \) nodes, for some integer constant \( φ \geq 1 \).

If \( p_i \) reaches the end of the list, it resets \( A[i] \) to \((\text{null}, ⊥)\) on line 0. On line 0, it again checks whether the second component of \( A[i] \) was \( ⊥ \) and, if so, returns the pointer that was in the first component of \( A[i] \). If no other process informed \( p_i \) before it reached the end of the list, then \( e \) is not in the list and \( p_i \) returns \( \text{NOT PRESENT} \).

Now suppose that, on line 0, \( p_i \) finds a node \( v \) containing \( e \) which is not at the front of the list. On line 0, it also checks whether the second component of \( A[i] \) is \( ⊥ \) and, if so, on lines 0 and line 0, resets \( A[i] \) to \((\text{null}, ⊥)\) and returns the pointer that was in the first component of \( A[i] \). Otherwise, on line 0, \( p_i \) sets the \( \text{old} \) field of its replacement node to point to node \( v \), indicating that \( v \) is an old node containing \( e \) which it is trying to replace. Then \( p_i \) tries to prepend its replacement node to the list.

To ensure that only one replacement for node \( v \) is prepended to the list, \( p_i \) first tries to change the compare-and-swap object \( v\.new \) from \( \text{null} \) to point to its replacement node in line 0. Then it sets \( g' \) to \( v\.new \) on line 0. If the CAS was successful, then \( g' \) points to its replacement node. If the CAS was unsuccessful, then either \( g' \) points to some other node, which is also a replacement for \( v \), or \( g' = \text{GONE} \), indicating that node \( v \) has already been removed from the list (and a replacement for \( v \) has already been prepended to the list). If \( g' \neq \text{GONE} \), then \( p_i \) tries to prepend the replacement node pointed to by \( g' \) to the list and remove \( v \) from the list by calling \( \text{MOVE-TO-FRONT}(g') \). In all cases, the first component of \( A[i] \) now points to a replacement for node \( v \) that has been prepended to the list. Then, on lines 0 and 0, \( p_i \) resets \( A[i] \) to \((\text{null}, ⊥)\) and returns the pointer that was in the first component of \( A[i] \).
In MOVE-TO-FRONT($g'$), $p_i$ repeatedly performs the following steps until the replacement node pointed to by $g'$ has been prepended to the list.

It first reads $Head$ on line 0 to get pointers, $h_1$ and $h_2$, to the first two nodes in the list. If the insertion of the first node is complete (i.e. $h_1.\text{old} = \text{DONE}$), then, on line 0, $p_i$ tries to prepend the node pointed to by $g'$ to the list by trying to change $Head$ from $(h_1, h_2)$ to $(w', h_1)$. Note that other processes may also be trying concurrently to prepend the same node or another node to the list. In all cases, $p_i$ then helps complete the insertion of the node at the front of the list.

To help complete the insertion of this node, $p_i$ first ensures that the first two nodes in the list point to one another by trying to set the $\text{next}$ field of the first node to point to the second on line 0 and trying to set the $\text{prev}$ field of the second node to point to the first on line 0. If either of these CAS operations is not successful, some other process did it first.

Then $p_i$ informs each other process $p_j$ that is currently looking for the item $e'$ in the node now at the front of the list. Specifically, for every announcement $A[j]$ that contains $e'$, a CAS is performed on line 0 to try to change it to $(h_1, \bot)$, provided $h_1$ still points to the front of the list. It is possible that $p_i$ could fall asleep for a long time between checking that $h_1$ still points to the front of the list and performing the CAS. In the meantime, it is possible that other nodes have been prepended to the list and $p_j$ has started another search for $e'$. In this case, the CAS should fail. This is why the announcement $A[j]$ contains a unique identifier for the search (which is a pointer to the replacement node allocated at the beginning of the search), in addition to the value being sought.

After this, the old copy of the new node at the front of the list is deleted from the list, by changing the $\text{next}$ field of its predecessor and the $\text{prev}$ field of its successor on lines 0 and 0. Note that if the old node was at the end of the list (i.e. its $\text{next}$ pointer is $\text{null}$), the second of these CAS operations is not performed. Finally, the $\text{new}$ pointer in this old copy is set to $\text{GONE}$ on line 0 and the $\text{old}$ pointer in the newly inserted node is changed to $\text{DONE}$ on line 0.

Note that when a node is removed from the list, a process that is traversing the list and is at that node will be able to continue traversing the list as if the node had not been removed. This is because the $\text{next}$ field of the removed node continues to point to the node that was its last successor.

The implementation can be made wait-free by using round-robin helping when trying to prepend a node to the list. Specifically, in addition to the pointers to the first two nodes in the list, $Head$ contains a modulo $p$ counter, $\text{priority}$, which indicates which process has priority for next prepending a node to the front of the list. Each time $Head$ is modified, the counter is incremented. Before trying to prepend the node $v'$ pointed to by $g'$, process $p_i$ checks $A[\text{priority}]$. If its second component is $\bot$, its first component contains a pointer to another node, and the $\text{old}$ field of that node is not yet $\text{DONE}$, then $p_i$ tries to prepend this other node instead of $v'$. Process $p_i$ repeats these steps at most $p$ times before node $v'$ is prepended.

### 6.1 Correctness

In this section, we prove that DMTF is \textit{linearizable}, which is a standard definition of correctness for distributed data structures. This means that, for every execution, it is possible to assign a distinct linearization point to every complete operation on the data structure and some subset of the incomplete operations such that the following two properties hold. First, the linearization point of each such operation occurs after it begins and, if it is complete, before it ends. Second, the result of every completed operation in the original execution is the same as in the corresponding
linearization, the execution in which the linearized operations are performed sequentially in order of their linearization points.

We begin by proving some observations and invariants about the old and new fields of nodes.

**Observation 17** The old field of a node only changes from NULL to point to a node that has been in the list and it only changes from pointing to a node to DONE. Once it is DONE, it never changes.

**Proof** The old field of a node is only changed on lines 0, which changes it from NULL to h, and 0, which changes it to DONE. Note that, before process p_i performs line 0, its local variable h is set to the second node in the list on line 0. It is only updated on line 0 by following next pointers. Since it is not NULL, by the test on line 0, h points to a node that has been in the list. It follows that the old field of a node never points to a node that has not been in the list.

**Observation 18** The new field of a node only changes from NULL to point to a node whose old field is not NULL and it only changes from pointing to a node to GONE. Once it is GONE, it never changes.

**Proof** The new field of a node is only changed on lines 0, which changes it from NULL to g, and 0, which changes it to GONE. Note that, before process p_i performs line 0, it changes g.old from NULL on line 0, if it has not already been changed. By Observation 17, the old field of a node never changes back to NULL. It follows that the new field of a node never points to a node whose old field is NULL.

**Observation 19** When MOVE-TO-FRONT(g') is called, g' points to a node whose old field is not NULL.

**Proof** By line 0, g' = h.new, where h is a local variable of process p_i that points to some node u. On line 0, the new field of u is changed from NULL to point to a node, if it has not already been changed. When p_i’s local variable g’ is set to h.new on line 0, Observation 18 implies that h.new is either GONE or points to a node whose old field is not NULL. By the test on line 0, g' ≠ GONE when MOVE-TO-FRONT(g') is called, so g' points to a node whose old field is not NULL.

**Lemma 20**

(a) The old field of a node that has not been in the list is either NULL or points to a different node containing the same item.

(b) The old field of the first node in the list is either DONE or points to a different node containing the same item.

(c) The old field of every node other than the first in the list and every node that is no longer in the list is DONE.

**Proof** by induction on the execution. Initially, this is true since the old field of every node in the list is DONE. When a node is newly allocated by a process p_i, its old field is set to NULL on line 0, which ensures that the claim continues to hold. Its old field is only changed on lines 0 and 0.

When the CAS on line 0 is successfully performed, the old field of the node v that is pointed to by p_i’s local variable g is changed from NULL to h. By Observation 17, h points to a node that has been in the list. By the test on line 0, h.item = e, which is the same as the item field of v. By the induction hypothesis, v has never been in the list. Hence, immediately after line 0, the old field of v points to a different node containing the same item and the claim continues to hold.
When the CAS on line 0 is successfully performed, it sets the old field of the node \( v \) to DONE, where \( v \) is the node pointed to by \( p_i \)'s local variable \( h_1 \). Note that \( h_1 \) was last set on line 0 to point to the first node in the list. Thus, immediately before line 0 is performed, \( v \) has been in the list, so by the induction hypothesis, its old field is not NULL. Thus the claim continues to hold.

Now, consider what happens when the node \( v \), pointed to by \( p_i \)'s local variable \( h' \), is prepended to the list (by a successful CAS on line 0). Immediately beforehand, the old field of the first node in the list is DONE, by lines 0 and 0 and the test on line 0. By Observation 19, the old field of \( v \) is not null and, by the test on line 0, it is not DONE. Thus, it points to some node \( u \). By the induction hypothesis, \( u \) and \( v' \) are different nodes containing the same item. When \( v \) is prepended to the list, the first node becomes the second node and \( v \) becomes the first node, so the claim continues to hold.

Finally, suppose \( p_i \) removes the node \( v' \) pointed to by its local variable \( h' \) from the list on line 0. When \( p_i \) performed line 0, it set its local variable \( h_1 \) to point to the first node in the list. When \( p_i \) performed line 0, it set its local variable \( h' \) equal to the old field of this node, which was either DONE or pointed to a different node containing the same item, by the induction hypothesis. By the test on line 0, \( h' \neq \) DONE. Thus, \( v' \) is not the first node in the list. By the induction hypothesis, the old field of \( v' \) is DONE. Hence, when \( v' \) is removed from the list, the claim continues to hold.

Lemma 21

(a) The new field of a node that has not been in the list is NULL.

(b) The new field of a node in the list is either NULL or points to a different node containing the same item.

(c) The new field of a node is GONE only after it has been removed from the list.

Proof by induction on the execution. Initially, this is true since the new field of every node is NULL. When a node is newly allocated by a process \( p_i \), its new field is set to NULL on line 0, which ensures that the claim continues to hold. Its new field is only changed on lines 0 and 0.

When the CAS on line 0 is successfully performed, \( h.new \) is changed from NULL to \( g \), which points to a node that has not yet been in the list. By line 0 and the test on line 0, \( g.item = e = h.item \). Since \( h \) points to a node in the list, \( h \) and \( g \) point to different nodes containing the same item. Thus the claim continues to hold.

If the CAS on line 0 is successfully performed, it sets the new field of the node \( u \) to GONE, where \( u \) is the node pointed to by \( p_i \)'s local variable \( h' \). It occurs after \( u \) has been removed from the list on lines 0–0. Thus the claim continues to hold.

Lemma 22 If the new field of a node \( u \) points to another node \( v \), then the old field of \( v \) points to \( u \).

Proof by induction on the execution. Initially, this is true since the new field of every node is NULL and the new field of every node is set to NULL on line 0 when it is allocated.

The new field of a node is only changed to point to a node by a successful CAS on line 0. Let \( u \) and \( v \) denote the nodes to which \( p_i \)'s local variables \( h \) and \( g \) point immediately before process \( p_i \) performs line 0. From the code, process \( p_i \) allocated \( v \) and is the only process that knows about this node. Thus, its CAS on line 0 successfully changes the old field of node \( v \) from NULL to point to node \( u \). If the CAS by process \( p_i \) on line 0 is also successful, then the new field of node \( u \) now points to \( v \).
When the old field of a node is changed from pointing to a node to done on line 0, the new field of the node it points to has already been changed to gone on line 0.

Next, we examine what happens when nodes are prepended and removed from the list.

**Observation 23** Immediately before a node is prepended to the list, the old field of the first node in the list is done.

**Proof** A node is prepended to the list on line 0. Process \( p_i \) sets its local variable \( h_1 \) to point to the first node, \( v \), in the list on line 0 and assigns \( h_1.\text{old} \) to its local variable \( h' \) on line 0. By the test on line 0, when \( p_i \) tries to prepend a node, \( h' = \text{DONE} \) and, by Observation 17, the old field of node \( v \) does not subsequently change. The CAS on line 0 is successful only if the first two nodes in the list have not changed since \( p_i \) last performed line 0. Hence, immediately before the node is prepended, \( v \) is the first node in the list and its old field is done.

**Observation 24** When a node is being removed from the list, the old field of the first node in the list points to it.

**Proof** A node is removed from the list on lines 0–0. Process \( p_i \) tries to remove a node in the list only when \( h' \neq \text{DONE} \), in which case, it tries to remove the node \( u \) to which \( h' \) points. Note that the old field of node \( v \) is only changed on line 0, after \( u \) has been removed by \( p_i \) (or some other process) on line 0. Thus, no other process can prepend a node to the list while a node is being removed from the list.

Thus, a node is not prepended to the list while another node is being removed and only one node is removed from the list at a time. By Lemma 20, the old field of a node never points to itself. Thus, when a node is removed, it is not the first node in the list.

**Lemma 25** No node is prepended to the list more than once.

**Proof** To obtain a contradiction, suppose there is a node that is prepended to the list more than once. Let \( v \) be the first node that is prepended to the list a second time and let \( p_i \) the process that successfully performs the CAS on line 0 to do this. By Observation 23, when it performs this line, the old field of the first node in the list is done. By Lemma 20, the old field of every other node that has been in the list is done. In particular, the old field of \( v \) is done. Thus, when \( p_i \) performs the test on line 0, \( g'.\text{old} = \text{DONE} \) and \( p_i \) does not perform the CAS. This is a contradiction.

**Lemma 26** If more than one process tries to remove a node from the list, the effect is the same as if only one process tries to remove it.

**Proof** When a node is allocated, its next field is initialized to null on line 0 and, when the node is first in the list, its next field is changed from null to point to the second node in the list on line 0. Subsequently, this field is only changed on line 0, when its successor is removed from the list. Likewise, the prev field of a node is initialized to null on line 0, when the node is second in the list, its next field is changed from null to point to the first node of the list in line 0, and thereafter, is only changed on line 0, when its predecessor is removed from the list.

By Observation 24, only the node, \( v \), pointed to by the old field of the first node in the list is removed. All processes trying to remove node \( v \) read its prev field to get a pointer to its predecessor (on line 0) and and its next field to get a pointer to its successor (on line 0). Note that once \( v \) has been removed from the list, its prev and next fields do not change. Processes use CAS to try to change \( v \)'s predecessor to point to \( v \)'s successor on line 0 and, if \( v \) is not the last node in the list, to try to change \( v \)'s successor to point to \( v \)'s predecessor on 0. The first such CAS operations are
successful, but subsequent ones are not, since these nodes no longer point to \( v \). Finally, processes use CAS to try to change the new field of \( v \) to GONE and the old field of the first node in the list to DONE. Since there are no steps that change a field with value GONE or DONE, only the first such CAS operations are successful.

**Lemma 27** Each item of the set has a node in the list that contains it. In every configuration, every node in the list whose old field is DONE contains a different item of the set.

**Proof** The proof is by induction on the execution. Initially, the claim is true, since the old field of every node in the list is DONE and the list consists of exactly one node for each item in the set.

By Observation 24, when a node is removed from the list, the old field of the first node in the list points to it. By Lemma 20, these two nodes contain the same item. Thus, after a node is removed, the claim continues to hold.

By Lemma 20, the old field of every node in the list is DONE, except possibly the first node. The old field of the first node in the list is changed to DONE on line 0 after the node to which it pointed has been removed from the list on lines 0–0. Thus, when the old field of a node is changed to DONE, the claim continues to hold.

By the test on line 0, the old field of a node being prepended is not DONE. The old field of a node is only changed to DONE on line 0 when the node is first in the list, so when the CAS on line 0 node is prepended, its old field is still not DONE. Thus, prepending a node does not change the set of nodes whose old field is DONE and the claim continues to hold.

Next, we consider how the announcement array can change.

**Observation 28**

(a) When \( A[i] = (\text{null}, \perp) \), process \( p_i \) can change its first component to the node it allocated at the beginning of its current search and its second component to the item it is searching for.

(b) When the second component of \( A[i] \) is an item, any process can change \( A[i] \) so that its second component is \( \perp \) and its first component points to a node that contains the item.

(c) When the first component of \( A[i] \) is not NULL, process \( p_i \) can change it to \( (\text{null}, \perp) \).

No other changes to \( A[i] \) are possible.

**Proof** From the code, process \( p_i \) can change \( A[i] \) from \( (\text{null}, \perp) \) on line 0, it can change \( A[i] \) to \( (\text{null}, \perp) \) on lines 0, 0, 0, 0, 0, and 0, and it can change \( A[j] \) for \( j \neq i \) on line 0. The CAS on line 0 changes \( A[j] \) to \( (h_1, \perp) \) only if its second component was \( e' \). By line 0, \( h_1 \) points to a node that contains the item \( e' \).

**Observation 29** If the second component of \( A[j] \) is \( e' \) when a node containing \( e' \) is prepended to the list, then it is changed to \( \perp \) before the other node in the list containing \( e' \) is removed from the list.

**Proof** When a node is prepended to the list, its old field is not DONE, by Lemma 20(a). By Observation 23 this field must be set to DONE before another node is prepended. From the code, before the old field of a node is set to DONE, which occurs on line 0, every announcement \( A[j] \) whose second component is \( e' \) is changed. This is because the CAS on line 0 is successful unless \( A[j] \) was changed since it was last read on line 0. By Observation 23 if \( A[j] \) was changed, its second component was changed to \( \perp \).
Lemma 30 When process $p_i$ performs line 0, the second component of $A[i]$ is $\perp$.

Proof Let $v$ be the node pointed to by $p_i$’s local variable $h$. By the test on line 0, $h$ points to a node, $u$, containing $e$. Note that $h$ was set to point to the second node in the list on line 0 and it was only updated on line 0 by following next pointers. Since nodes are only prepended to the list, $u$ was in the list when $p_i$ performed line 0. By the test on line 0, the node at the front of the list when $p_i$ performed line 0 does not contain $e$. On line 0, $p_i$’s local variable $g'$ is set to $u$’s new field, which, by the CAS on line 0, is not NULL.

If $g'$ is gone, then, by Lemma 21, $u$ has been removed from the list. Otherwise, by Lemmas 21 and 22, $u$’s new field points to a different node $v$, whose old field points to $u$. In this case, $p_i$ calls MOVE-TO-FRONT($g'$), where $g'$ points to $v$. By the test on line 0, $v$’s old field is done when $p_i$ returns from MOVE-TO-FRONT. Thus, by Lemmas 22 and 18, $u$’s new field no longer points to $v$ and, hence is gone. Lemma 21 implies that $u$ has been removed from the list. In both cases, it follows from Observation 29 that the second component of $A[i]$ has been changed to $\perp$. By Observation 28, no other process changes $A[i]$ if its second component is $\perp$. Therefore, when process $p_i$ performs line 0, the second component of $A[i]$ is $\perp$. □

Lemma 31 When process $p_i$ performs line 0, the second component of $A[i]$ is $\perp$ if the item it is searching for is in the list.

Proof Suppose item $e$ is in the set and $p_i$ performs line 0 during an invocation of SEARCH($e$). Lemma 27 implies that there was a node $v$ containing $e$ in the list when $p_i$ performed line 0. When $p_i$ performed line 0, the node at the beginning of this list did not contain $e$, otherwise $p_i$ would have returned on line 0. Since $p_i$ reached the end of the list without finding a node containing $e$, node $v$ must have been removed from the list after $p_i$ announced its search on line 0 and before it read $A[i]$ on line 0. It follows from Observation 29 that the second component of $A[i]$ was changed to $\perp$. By Observation 28, no other process changes $A[i]$ if its second component is $\perp$. From the code, $p_i$ does not change $A[i]$ back to (NULL, $\perp$) before it performs line 0. Thus, when process $p_i$ performs line 0, the second component of $A[i]$ is $\perp$. □

Lemma 32 When process $p_i$ is not performing SEARCH, $A[i] = (\text{NULL}, \perp)$. When $A[i] = (\text{NULL}, \perp)$, either $p_i$ is not performing SEARCH or has not yet performed line 0 during its current invocation of SEARCH.

Proof The proof is by induction on the execution. $A[i]$ is initially (NULL, $\perp$) and $p_i$ is not performing SEARCH. We consider each step in the execution where $A[i]$ might change.

Before $p_i$ performs line 0 during an invocation of SEARCH($e$), $A[i] = (\text{NULL}, \perp)$, by the induction hypothesis. Thus, the CAS on this line successfully changes $A[i]$ so that its second component is $e \neq \perp$.

On line 0, $h_1$ is a pointer to the node at the front of the list by line 0 and $b$ is an item by the test on line 0. Thus, this step does not change $A[i]$ to or from (NULL, $\perp$).

The only other lines in which $A[i]$ is changed are 0, 0, 0, 0, 0, and 0. In all these lines, $p_i$ uses a CAS to try to change $A[i]$ to (NULL, $\perp$). With the exception of the CAS on line 0, process $p_i$ returns immediately following each of those lines. However, if the CAS on line 0 is successful, then the CAS on line 0 is not performed, so $p_i$ also returns immediately in this case. Thus, $p_i$ returns from SEARCH immediately after $A[i]$ is set to (NULL, $\perp$).

It remains to show that when $p_i$ returns from SEARCH, $A[i] = (\text{NULL}, \perp)$. Note that $p_i$ changes $A[i]$ from (NULL, $\perp$) exactly once during each invocation of SEARCH, when it performs line 0. By
Observation 28 no other process changes \( A[i] \) from (or to) \((\text{null}, \perp)\). Thus, it suffices to show that \( p_i \) changes \( A[i] \) to \((\text{null}, \perp)\) between when it performs line 0 and it returns from SEARCH.

First suppose that \( p_i \) returns from SEARCH on line 0. Prior to this, it performs the CAS on line 0. If this CAS was successful, \( A[i] \) is changed to \((\text{null}, \perp)\). If this CAS was unsuccessful, \( A[i] \) was changed by another process since \( p_i \) set it to \((w, e)\) on line 0. By Observation 28, the second component of \( A[i] \) was changed to \( \perp \) and \( A[i] \) does not change again until \( p_i \) changes it. Hence \( p_i \) performs the CAS on line 0, which successfully changes \( A[i] \) to \((\text{null}, \perp)\).

The only other lines on which \( p_i \) returns from SEARCH are 0, 0, 0, and 0. Immediately prior to performing any of these lines, \( p_i \) reads \( A[i] \) and then performs a CAS. If no other process changes \( A[i] \) between these two steps, the CAS sets \( A[i] \) to \((\text{null}, \perp)\). By Observation 28 no other process changes \( A[i] \) if its second component is \( \perp \).

Immediately prior to when \( p_i \) returns from SEARCH on line 0, it performs the CAS on line 0. By the test on line 0, the second component of \( A[i] \) was \( \perp \) when \( p_i \) read \( A[i] \) on line 0. Thus the CAS successfully changes \( A[i] \) to \((\text{null}, \perp)\).

Similarly, immediately prior to when \( p_i \) returns from SEARCH on line 0, it performs the CAS on line 0. By the test on line 0, the second component of \( A[i] \) was \( \perp \) when \( p_i \) read \( A[i] \) on line 0, so the CAS successfully changes \( A[i] \) to \((\text{null}, \perp)\).

Now, suppose \( p_i \) returns from SEARCH\((e)\) on line 0. By Lemma 30 when \( p_i \) performed line 0, the second component of \( A[i] \) was \( \perp \). By Observation 28 no other process changes \( A[i] \) if its second component is \( \perp \). Hence, after the CAS on line 0, \( A[i] = (\text{null}, \perp)\).

Finally, suppose that \( p_i \) returns from SEARCH on line 0. When \( p_i \) performs the CAS on line 0, it has reached the end of the list without finding a node with item \( e \). If \( e \) is not in the set, then Observation 28 implies that no other process can change \( A[i] \) from \((w, e)\) to anything else. If \( e \) is in the set, then, by Lemma 31 the second component of \( A[i] \) was \( \perp \) when process \( p_i \) performed line 0. By Observation 28 \( A[i] \) was not changed by any other process between when \( p_i \) performed lines 0 and 0. Thus, in both cases, the CAS on line 0 successfully changes \( A[i] \) to \((\text{null}, \perp)\).

**Lemma 33** Suppose a process invokes SEARCH\((e)\), where \( e \) is not an item in the set. If the process does not crash, then it returns NOT PRESENT.

**Proof** Suppose \( p_i \) invokes SEARCH\((e)\), where \( e \) is not an item of the set and, hence, is not contained in any node in the list. While \( p_i \) is performing SEARCH\((e)\), its tests on lines 0 and 0 are never successful. Thus \( p_i \) does not return on line 0, 0, or 0. Moreover, \( p_i \) does not call MOVE-TO-FRONT. By Lemma 22 \( A[i] \neq (\text{null}, \perp) \) between the step in which \( p_i \) sets the second component of \( A[i] \) to \( e \) on line 0 and the step in which it returns. By Observation 28 no other process changes \( A[i] \) when its second component is \( e \). Hence, \( p_i \)'s test on line 0 is never successful while \( p_i \) is performing SEARCH\((e)\) and, so, \( p_i \) does not return on line 0.

Therefore, in each iteration of the loop, \( p_i \) updates \( h \) on line 0 to point to the next node in the list. Since \( h \) was set to point to the second node in the list on line 0 and nodes are only prepended to the list, eventually the end of the list is reached and \( h = \text{null} \). The second component of \( A[i] \) is still \( e \) when \( p_i \) performs line 0. Thus, SEARCH\((e)\) returns NOT PRESENT on line 0.

If \( e \) is not an item in the set, then SEARCH\((e)\) does not modify the list. Hence, this operation can be linearized at any point during its operation interval, for example, when it returns. (Note that this case is not an option in the problem on which we perform a competitive analysis. Both
Lemma 34 If process $p_i$ returns from SEARCH($e$) on line 0, then $p_i$ returns a pointer to a node containing $e$, which was at the front of the list when $p_i$ read Head on line 0.

Proof Suppose process $p_i$ returns from SEARCH($e$) on line 0. Then, as a result of performing line 0, $h_1$ is set to point to the first node in the list. By the test on line 0, this node contains $e$. Then it returns $h_1$ on line 0.

In this case, the instance of SEARCH($e$) by process $p_i$ is linearized when $p_i$ read Head on line 0.

Lemma 35 If $A[j]$ is changed to $(h_1, \bot)$, then the node to which $h_1$ points was at the front of the list at some time since $p_j$ announced its current search.

Proof Suppose $A[j]$ is changed from $(a, b)$ to $(h_1, \bot)$ by processor $p_j$. By Observation 28, $b$ is the item $p_j$ is currently searching for, $a$ is a pointer to the node that $p_j$ allocated at the beginning of this search, and $A[j]$ has not changed since $p_j$ announced its current search on line 0. By line 0, $p_j$’s local variable $h_1$ points to a node $v$ that contains the item $e'$. When $p_i$ last performed line 0, $v$ was at the front of the list. After checking that $b = e'$, $p_i$ checks that $h_1.\text{old} \neq \text{DONE}$, which, by Lemma 20 implies that $v$ is still at the front of the list.

Lemma 36 If $e$ is in the list and process $p_i$ returns from SEARCH($e$) on line 0, then $p_i$ returns a pointer to a node containing $e$, which was at the front of the list at some time since $p_i$ announced this search on line 0.

Proof By Lemma 31 the second component of $A[i]$ was $\bot$ when process $p_i$ performed line 0. From the code, $p_i$ has not tried to change $A[i]$ to (null, $\bot$) since it changed it from (null, $\bot$) on line 0. By Observation 28, it follows that $A[i] = (h_1, \bot)$, where $h_1$ points to a node, $v$, that contains $e$. By the test on line 0, $p_i$ returns $h_1$. By Lemma 35 when $A[i]$ was changed to $(h_1, \bot)$, $v$ was at the front of the list. Thus, $p_i$ returns a pointer to a node containing $e$, which was at the front of the list at some time since $p_i$ announced this search on line 0.

Lemma 37 If process $p_i$ returns from SEARCH($e$) on line 0, 0, or 0, then $p_i$ returns a pointer to a node containing $e$, which was at the front of the list at some time since $p_i$ announced this search on line 0.

Proof If $p_i$ returns from SEARCH($e$) on line 0, 0, or 0, it has performed line 0, 0, or 0. If $p_i$ performs the CAS on line 0, then by the test on line 0, the second component of $A[i]$ was $\bot$ when $p_i$ read $A[i]$ on line 0. If $p_i$ performs the CAS on line 0, then by the test on line 0, the second component of $A[i]$ was $\bot$ when $p_i$ read $A[i]$ on line 0. By Lemma 30 when process $p_i$ performs line 0, the second component of $A[i]$ is $\bot$. By Observation 28 no other process changes $A[i]$ if its second component is $\bot$. Hence, immediately before process $p_i$ performs the CAS on line 0, 0, or 0, the second component of $A[i]$ is $\bot$. From the code, $p_i$ has not tried to change $A[i]$ to (null, $\bot$) since it changed it from (null, $\bot$) on line 0. By Observation 28 it follows that $A[i] = (h_1, \bot)$, where $h_1$ points to a node, $v$, that contains $e$. By Lemma 35 when $A[i]$ was changed to $(h_1, \bot)$, $v$ was at the front of the list. Thus, $p_i$ returns a pointer to a node containing $e$, which was at the front of the list at some time since $p_i$ announced this search on line 0.

In these remaining cases, the instance of SEARCH($e$) by process $p_i$, which returns a pointer to a node $v$ containing $e$, can be linearized at any point which is after $p_i$ announces this search on line 0, before $p_i$ returns from the search, and at which $v$ was at the front of the list.
Thus, we have shown that the algorithm, DMTF, presented in Figures 1 and 2 is correct.

**Theorem 38** When a SEARCH(e) operation is linearized, a node containing e is at the front of the list and the operation returns a pointer to this node, if e is in the set. Otherwise, the operation returns NOT PRESENT.

7 Competitive Analysis of DMTF– Fully Adversarial

The analysis at the item access level counts the number of accesses performed per request. It ignores all costs not directly associated with the search.

In DMTF, each item accessed in the list involves a constant number of shared memory operations. Furthermore, a process does an additional check of its announcement array once every $\phi$ nodes. Thus, the cost of a request is an $O(1 + 1/\phi)$ factor more than the number of item accesses performed to handle the request.

There is at most one move-to-front operation per request. Each successfully completed move-to-front would take $\Theta(p)$ steps if it was performed sequentially: a constant number of updates of fields in nodes and $\Theta(p)$ steps for informing the other processes. However, in a distributed execution, it is possible that all $p$ processes help perform the move-to-front. Thus, each request contributes $O(p^2)$ to the cost. There is $O(\phi)$ extra cost per request for the nodes a process accesses in the list after it has been informed that a node containing the item it is searching for has been moved to the front of the list. Finally, because the item in the node at the front of the list can occur in another node of the list, there is an $O(1)$ additional cost.

Since OPT’s cost is at least 1 for each request, it follows from Theorem 16 that, for the fully adversarial scheduler, if $\phi \in O(p^2)$, then

$$\text{DMTF}(I) \leq O(p^2) \text{OPT}(I) + O(1),$$

where the additive constant depends only on $p$, $\ell$, and $\phi$. The lower bound of Theorem 6 shows that any distributed algorithm (even one which treats the requests in an optimal manner sequentially), must have

$$\text{DMTF}(I) \geq (2p^2 - p) \text{OPT}(I) - O(1),$$

if request sequences can be merged arbitrarily. Therefore, the cost of merging of the sequences dominates the other costs and

$$\text{DMTF}(I) = \Theta(p^2) \text{OPT}(I) + O(1).$$

8 A Linearization-Based Analysis

At the operation level, since the comparison is made on the same sequences, the classic result from online algorithms gives the ratio $2 - \frac{2}{\ell+1}$ [27]. We proceed to the item access level.

Consider any execution $\epsilon$ of DMTF on the request sequences $\sigma_1, \sigma_2, \ldots, \sigma_p$. Let DMTF($\epsilon$) denote the cost of $\epsilon$ at the item access level. Let LIN$_{\epsilon}(\sigma_1, \sigma_2, \ldots, \sigma_p)$ denote the sequence of requests served in some linearization of $\epsilon$ and let OPT(LIN$_{\epsilon}(\sigma_1, \sigma_2, \ldots, \sigma_p)$) denote the cost of an optimal sequential execution on LIN$_{\epsilon}(\sigma_1, \sigma_2, \ldots, \sigma_p)$.
Note that MTF’s cost on LIN_(σ₁, σ₂, ..., σₚ) could be much less than DMTF(ε): When an item x is requested k times in a row in LIN_(σ₁, σ₂, ..., σₚ), DMTF could have had up to min{p, k} processes concurrently searching for x in the list. Those processes could each incur cost equal to the index, i, that x had in the list. In the case where the move-to-front occurs before the other processes begin searching for x, the first process incurs cost i and the remainder incur cost 1.

To compare DMTF(ε) and OPT(LIN_(σ₁, σ₂, ..., σₚ)), we use the list factoring and phase partitioning techniques as discussed in [20], using the partial cost model. With list factoring, each distinct pair of items, x and y, in the original list, L, is considered separately. They are considered in a list, Lₓ,y, containing only these two items (in the same order as in L at any point in time) and with the subsequence of requests, LIN_(σ₁, σ₂, ..., σₚ)ₓ,y, to these items that occur in LIN_(σ₁, σ₂, ..., σₚ) (also in the same order).

The pairwise property says that the items x and y are in the same order with respect to each other in L at every point during the request sequence LINₐ(σ₁, σ₂, ..., σₚ) as they are in Lₓ,y at corresponding point in the request sequence LINₐ(σ₁, σ₂, ..., σₚ)ₓ,y. The pairwise property holds for MTF because, whenever a request to an item in the list is served, the item is moved to the front of the list. It holds for DMTF because, in addition, each request to an item in the list is linearized when the item is at the front of the list. In the partial cost model, the cost of every search is one less than in the full cost model; only unsuccessful item inspections are counted. Cost independence means that the algorithm makes decisions regardless of the cost it pays, i.e., it behaves the same under all cost models. DMTF is cost independent.

The list factoring and phase partitioning techniques [19, 27, 32, 4, 2] have become standard in studying the list accessing problem. The following results are well known (see [20]). Consider the execution εₓ,y obtained from ε by removing all steps by processes while they are not performing SEARCH(x) or SEARCH(y) and all accesses to nodes except those containing x or y. Note that DMTF does not use paid exchanges and has the pairwise property. Thus, if we can show that, for every sequence and for every pair of items, x and y, the partial cost of εₓ,y is at most c times the partial cost of OPT on LINₐ(σ₁, σ₂, ..., σₚ)ₓ,y, then the partial cost of ε is at most c times the partial cost of OPT on LINₐ(σ₁, σ₂, ..., σₚ). Since DMTF is cost independent, it follows that DMTF(ε) ≤ c OPT(LINₐ(σ₁, σ₂, ..., σₚ)).

Recall that, with item access level interleaving, we only consider the costs of accessing nodes while searching through the list, with total cost i for an item in location i of the list.

**Theorem 39** With respect to item access level interleaving in the linearization-based model, for any execution, ε, of DMTF on the request sequences σ₁, σ₂, ..., σₚ,

\[ \text{DMTF}(ε) \leq (p + 1) \text{OPT}(\text{LIN}_ε(σ₁, σ₂, ..., σₚ)) + O(1), \]

where the constant depends only on p and ℓ.

**Proof** We partition LIN_(σ₁, σ₂, ..., σₚ)ₓ,y into phases which are defined inductively as follows. Assume that, for some t ≥ 1, we have defined phases up until, but not including, the t’th request and the relative order of the two items in OPT’s list is x, y before the t’th request. Then the next phase is of type 1 and is of one of the following forms, where j ≥ 0 and k ≥ 1:

(a) yyy^j \quad (b) (yx)^k yyy^j \quad \text{and} \quad (c) (yx)^k x x^j.

Phases continue until the next request to a different item (to x in forms (a) and (b), and to y in form (c)). There may be one incomplete phase for each pair of items, but this only adds a constant
to the costs. Note that this phase partitioning is different than the most common type, since we do not start a new phase after only two identical requests, but wait until we get a different request. This is necessary for the analysis, since processes may be searching for the same item at the same time.

In case the relative order of the items is $y,x$ before the $t$'th request, the phase has type 2 and its form is exactly the same as above with $x$ and $y$ interchanged.

Table 2 shows the costs incurred by the two algorithms, and the worst case ratio, for each of the three forms of type 1. Note that the same results hold for type 2 forms.

| Phase   | DMTF | OPT | Ratio  |
|---------|------|-----|--------|
| $yyyy^j$ | $\leq p$ | 1   | $\leq p$ |
| $(yx)^kyyyy^j$ | $\leq 2k + p$ | $k + 1$ | $\leq 2 + \frac{p-2}{k+1}$ |
| $(yx)^kxx^j$ | $\leq 2k + p - 1$ | $k$ | $\leq 2 + \frac{p-1}{k}$ |

Table 2: The costs of DMTF and OPT under the partial cost model for a phase of type 1 (i.e., the initial ordering of items is $x,y$) and the maximum ratio of these costs.

The column for DMTF holds because, in phases of form (a), at most $p$ processes would find $y$ at index 2 (and thus have cost 1 in the partial cost model); in phases of form (b) and (c), DMTF has partial cost 1 for each of the alternating occurrences of $x$ and $y$, plus at most partial cost $p$ for the requests to the same item at the end.

For form (a), OPT moves $y$ to the front immediately, so it only has cost 1. For forms (b) and (c), OPT does not do any moves while the $x$ and $y$ are alternating, but in form (b), it moves the first of the $y$'s after the alternation to the front. Thus, it has cost $k + 1$ for form (b) and $k$ for form (c).

The maximum ratio of DMTF’s to OPT’s performance is, thus, bounded by $\max\{p, 2 + \frac{p-1}{k}\}$. As mentioned above, since DMTF is cost independent, this bound also holds in the full cost model. The term $p$ in $\max\{p, 2 + \frac{p-1}{k}\}$ is the dominating term as long as $p \geq 3$ and $k \geq 2$. If $p = 2$, the larger term is $2 + \frac{1}{k} \leq 3$. If $p = 1$, then the result is the standard 2-competitiveness of MTF. If $k = 1$, then the result is $2 + p + 1$. Thus, one concludes that, for any execution $\epsilon$, $\text{DMTF}(\epsilon) \leq (p + 1) \text{OPT}(\text{LIN}\epsilon(\sigma_1,\sigma_2,\ldots,\sigma_p))$. $\square$

At the level of the actual algorithm, DMTF, the analysis from the fully adversarial model shows that the cost of each search only increases by an $O(1+1/\phi)$ factor and an additive $O(p^2 + \phi)$ term. Thus, when $\phi \in O(p)$ and OPT's average cost per request is $\Omega(p)$, DMTF is $\Theta(p)$-competitive.

9 Concluding Remarks

The List Accessing problem is the first self-adjusting data structure problem considered in a distributed setting, where the processes each have their own request sequence, and a competitive analysis is performed. It seems reasonable to assume that the concerns about the power of the scheduler, which have arisen for the list accessing problem, would also arise for other distributed data structure problems, where the individual processes have their own request sequences.

We have presented two different models for performing competitive analysis in a distributed setting.
More online problems in a distributed setting should be investigated to determine how one best assesses the quality of such algorithms. In that context, it is interesting to know what the effect of the scheduler is.

References

[1] Miklós Ajtai, James Aspnes, Cynthia Dwork, and Orli Waarts. A theory of competitive analysis for distributed algorithms. In 35th Annual Symposium on Foundations of Computer Science (FOCS), pages 401–411. IEEE Computer Society, 1994.

[2] Susanne Albers. Improved randomized on-line algorithms for the list update problem. SIAM J. Comput., 27(3):682–693, 1998.

[3] Susanne Albers and Dario Frascaria. Quantifying competitiveness in paging with locality of reference. In 42nd International Colloquium on Automata, Languages, and Programming (ICALP), volume 9134 of Lecture Notes in Computer Science, pages 26–38. Springer, 2015.

[4] Susanne Albers, Bernhard von Stengel, and Ralph Werchner. A combined BIT and TIMES-TAMP algorithm for the list update problem. Inform. Process. Lett., 56:135–139, 1995.

[5] Dan Alistarh, James Aspnes, Michael A. Bender, Rati Gelashvili, and Seth Gilbert. Dynamic task allocation in asynchronous shared memory. In Proceedings of the twenty-fifth annual ACM-SIAM Symposium on Discrete Algorithms, (SODA), pages 416–435, 2014.

[6] Noga Alon, Bil Kalai, Moty Ricklin, and Larry Stockmeyer. Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling. Theor. Comput. Sci., 130:175–201, 1994.

[7] Spyros Angelopoulos and Pascal Schweitzer. Paging and list update under bijective analysis. J. ACM, 60(2):7:1–7:18, 2013.

[8] James Aspnes. Competitive analysis of distributed algorithms. In Amos Fiat and Gerhard J. Woeginger, editors, Online Algorithms, The State of the Art, volume 1442 of Lecture Notes in Computer Science, pages 118–146. Springer, 1998.

[9] James Aspnes and William Hurwood. Spreading rumors rapidly despite an adversary. J. Algorithms, 26(2):386–411, 1998.

[10] James Aspnes and Orli Waarts. Compositional competitiveness for distributed algorithms. J. Algorithms, 54(2):127–151, 2005.

[11] Hagit Attiya and Jennifer Welch. Distributed Computing: Fundamentals, Simulations, and Advanced Topics. Wiley-Interscience, 2 edition, 2004.

[12] Baruch Awerbuch and Yossi Azar. Local optimization of global objectives: Competitive distributed deadlock resolution and resource allocation. In 35th Annual Symposium on Foundations of Computer Science (FOCS), pages 240–249. IEEE Computer Society, 1994.

[13] Baruch Awerbuch, Yair Bartal, and Amos Fiat. Distributed paging for general networks. J. Algorithms, 28:67–104, 1998.

[14] Baruch Awerbuch, Yair Bartal, and Amos Fiat. Competitive distributed file allocation. Inform. Comput., 185(1):1–40, 2003.
[15] Baruch Awerbuch, Shay Kutten, and David Peleg. Competitive distributed job scheduling (extended abstract). In 24th Annual ACM Symposium on Theory of Computing (STOC), pages 571–580. ACM, 1992.

[16] Baruch Awerbuch and Adi Rosén. The distributed $k$-server problem — a competitive distributed translator of $k$-server algorithms. *J. Algorithms*, 28:67–104, 1998.

[17] Yair Bartal, Amos Fiat, and Yuval Rabani. Competitive algorithms for distributed data management. *J. Comput. Syst. Sci.*, 51(3):341–358, 1995.

[18] Shai Ben-David, Allan Borodin, Richard M. Karp, Gábor Tardos, and Avi Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11(1):2–14, 1994.

[19] Jon Louis Bentley and Catherine C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Commun. ACM*, 28:404–411, 1985.

[20] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[21] Joan Boyar, Lene M. Favrholdt, Christian Kudahl, Kim S. Larsen, and Jesper W. Mikkelsen. Online algorithms with advice: A survey. *ACM Comput. Surv.*, 50(2):19:1–19:34, 2017.

[22] Joan Boyar, Shahin Kamali, Kim S. Larsen, and Alejandro López-Ortiz. On the list update problem with advice. *Inform. Comput.*, 253:411–423, 2017.

[23] Reza Dorrigiv, Martin R. Ehmsen, and Alejandro López-Ortiz. Parameterized analysis of paging and list update algorithms. In Evripidis Bampis and Klaus Jansen, editors, *Approximation and Online Algorithms, 7th International Workshop, (WAOA)*, volume 5893 of *Lecture Notes in Computer Science*, pages 104–115. Springer, 2010.

[24] Martin R. Ehmsen, Jens S. Kohrt, and Kim S. Larsen. List factoring and relative worst order analysis. *Algorithmica*, 66(2):287–309, 2013.

[25] Esteban Feuerstein and Alejandro Strejilevich de Loma. On-line multi-threaded paging. *Algorithmica*, 32(1):36–60, 2002.

[26] Esteban Feuersteina, Steven S. Seiden, and Alejandro Strejilevich de Loma. On multi-threaded metrical task systems. *J. Discrete Algorithms*, 4(3):401–413, 2006.

[27] Sandy Irani. Two results on the list update problem. *Inform. Process. Lett.*, 38(6):301–306, 1991.

[28] Tracy Kimbrel. Interleaved prefetching. *Algorithmica*, 32:107–122, 2002.

[29] Bruce M. Maggs, Friedhelm Meyer auf der Heide, Berthold Vöcking, and Matthias Westermann. Exploiting locality for data management in systems of limited bandwidth. In 38th Annual Symposium on Foundations of Computer Science (FOCS 1997), pages 284–293. IEEE Computer Society, 1997.

[30] Rahul Simha and Amitava Majumdar. On lookahead in the list update problem. *Inform. Process. Lett.*, 50(2):105–110, 1994.

[31] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.
[32] Boris Teia. A lower bound for randomized list update algorithms. *Inform. Process. Lett.*, 47:5–9, 1993.