Competitive Concurrent Distributed Data Structures
(Draft: Do not distribute)

Baruch Awerbuch Yair Bartal Amos Fiat Rainer Gawlick

November 29, 1994

1 Introduction

The standard way to measure performance of *centralized online* algorithms is to consider the "competitive ratio" which is the worst-case performance ratio between online and optimum offline algorithms on a specific input instance. In such algorithms decisions are made by *global* controller, that has full information about past input. This model, pioneered by Sleator and Tarjan [ST85] is used by most of the previous work in this area, e.g. [BBK+90, MMS88, FKL+88, KP94].

In contrast, *concurrent distributed* algorithms are ones where decisions are made by in a decentralized manner, i.e. each component of the system makes an independent decision, and many new inputs can come simultaneously. This notion was introduced by Deng and Papadimitriou [XP92, PY93] in context of one-shot multi-player games and by Awerbuch, Kutten and Peleg [AKP92] in the context of dynamic job scheduling. In the context of dynamically changing networks, such issues were analyzed by Awerbuch and Leighton [AL94]. In the context of asynchronous memory systems, this was studied by Awerbuch and Azar and Ajtai et el [AADW94].

We comment that concurrent distributed directory is a central problem in maintaining Virtual Shared memory in current parallel multiprocessor architectures [ALKK88, CFKA90, JLGS90, LEH85, LLG+90]. Certainly, in these settings the issues of asynchronicity and concurrency cannot be ignored.

Previously, distributed directory has only been considered in the setting where all the operations occur in a serial order [BFR92]; direct applications of methods in [BFR92] in concurrent setting lead to competitive ratio which grows *lenearly* with the number of network noded.

The contributions of this paper are

- definition of semantics and complexity measures for distributed data structure for concurrent asynchrnous distributed directory access.
- implementation of concurrent asynchronous distributed directory supporting Insert's and Find's with poly-logarithmic overhead using the techniques in [BFR92] with additional synchronoization mechanisms.

2 Problem statement.

2.1 Network Model

Consider an asynchronous distributed a network described by an undirected graph $G(V, E, w)$ consting of nodes v, edges E, and positive edge weight function w. Messages sent over network edges arrive within some finite yet undetermined time [Gal82, Awe85].

We assume the existence of a *weight* function $w : E \to \mathcal{R}^+$, assigning an arbitrary positive weight $w(e)$ to each edge $e \in E$. For two vertices v, u in a graph G let $dist_G(v, u)$ denote the (weighted) length of a shortest path in G between those vertices, i.e., the cost of the cheapest path connecting them, where the cost of a path $(e_1, \ldots, e_s)$ is $\sum_{1 \le i \le s} w(e_i)$. (We usually omit the subscript G where no confusion arises.)

The communicaiton cost of a protocol is the total cost of all messages transmitted, where each message can carry logarithmic number of bits, and cost of message transmission over an edge e is the weight $w(e)$ of that edges.

2.2 Semantics

The Basic Distributed Directory (BDD) is a distributed data structure supporting the operations `find` and `insert` on dynamically growing set $\mathcal{S}$ of network nodes. The semantics of these operations are

- `find` [node in the set $u \in \mathcal{S}$] [from arbitrary node $v \in V$]: this operation, called from some arbitrary node $v \in V$ should return name of a node $u \in S$.
- `insert` [arbitrary node $v \in V$ to set $\mathcal{S}$]: adds new node v to set $\mathcal{S}$.

2.3 Complexity for serial executions

We defined the complexity measures as follows. Let $\mathcal{F}$ be the set of all find operations and let $F \in \mathcal{F}$ be

The *competitiveness* of `find` operation is the ratio

$$\underset{F}{ratio} = \max_{F \in \mathcal{F}} \frac{cost\ F}{dist(F)}$$

the communication cost of that operation, divided by the distance from a new user to a previous user or a source.

It is not difficult to implement such operations by "brute force", namely broadcasting search message thru the whole network in the case of `Find`, and/or broadcasting an update thru the whole network in case of `Insert` or `Delete`. Another possibility is having a central network controlller.

2.4 Complexity Measures (Distributed concurrent Competitiveness)

To model concurrent executions, we divide operations such as a Read or a Write operation into a sequence of atomic steps. A *concurrent execution* is just a sequence of atomic steps where the atomic steps of concurrent operations are interleaved. The operations in a concurrent execution α are *atomic* if the following condition holds. It must be possible to associate each operation in α with a single point, called a *serialization point*, between the first and the last atomic steps of the operation such that the responses of the operations in α could be the responses if the operations in α were executed serially based on the serial order implied by the serialization points. The serial order implied by the serialization points of a concurrent executions is called the *serialization order*. In general, a concurrent execution can have many serialization orders. The serialization orders of a concurrent execution α are denoted by $s(\alpha)$. [[[This section needs the appropriate references to past work.]]] [[[Perhaps concepts such as well formedness should also be introduced.]]]

We are now ready to give a precise definition of the complexity measures that we use.

The cost of transmitting an arbitrary message from node v to node u is $dist(v, u)$. [[[Note that the cost may increase when the message contains a lot of data.]]]

Now consider an algorithm A that solves some problem P. Let α be an execution of A. The costs incurred by execution α, denoted by $cost(\alpha)$, is the sum of the costs of all messages sent in α. Now consider any algorithm B that is not necessarily concurrent and that solves P. For any serialization order $\gamma \in s(\alpha)$, let $cost_B(\gamma)$ be the cost of the serial execution by algorithm B of the operations in γ in the order given by γ. Let $opt(\gamma) = min_B\{cost_B(\gamma)\}$. Now define $opt(\alpha) = max_{\gamma\in s(\alpha)}(opt(\gamma))$ We say that $opt(\alpha)$ is the optimal cost of α. Now we define the *competitive factor* of a algorithm A to be:

$$CF(A) = \max_{\alpha} \left(\frac{cost(\alpha)}{opt(\alpha)} \right).$$

[[[The justification for this definition still needs to be written down.]]]

3 Find, Copy, Delete, and Modify Primitives

This section introduces the algorithm, that implements the `find` and `copy` primitives. The algorithm is distributed and concurrently competitive with a polylogarithmic competitive factor. The algorithm is based on the regional covers defined in [?].

3.1 Preliminaries

3.1.1 Graph Theory

Next let us define some basic graph notation. The *d-neighborhood* of a vertex $v \in V$ is defined as $N_d(v) = \{u \mid dist(v,u) \le d\}$. Given a subset of vertices $R \subseteq V$, denote $\mathcal{N}_m(R) = \{N_m(v) \mid v \in R\}$. Let $D = Diam(G)$ denote the *diameter* of the network, i.e., $max_{v,u \in V}(dist(v,u))$. For a vertex $v \in V$, let $Rad(v,G) = max_{v \in V}(dist_G(v,u))$. Let $Rad(G)$ denote the *radius* of the network, i.e., $min_{v \in V}(Rad(v,G))$. A *center* of G is any vertex v realizing the radius of G (i.e., such that $Rad(v,G) = Rad(G)$. In order to simplify some of the following definitions we avoid problems arising from 0-diameter or 0-radius graphs, by defining $Rad(G) = Diam(G) = 1$ for the single-vertex graph $G = (\{v\}, \emptyset)$. Observe that for every graph G, $Rad(G) \le Diam(G) \le 2\,Rad(G)$. (Again, in all of the above notations we usually omit the reference to G where no confusion arises.)

Finally, let us introduce some definitions concerning covers. Given a set of vertices $S \subseteq V$, let $G(S)$ denote the subgraph induced by S in G. A *cluster* is a subset of vertices $S \subseteq V$ such that $G(S)$ is connected. We use $Rad(v,S)$ (respectively, $Rad(S)$, $Diam(S)$) as a shorthand for $Rad(v, G(S))$ (resp., $Rad(G(S))$, $Diam(G(S))$). A *cover* is a collection of clusters $\mathcal{S} = \{S_1, \ldots, S_m\}$ such that $\bigcup_i S_i = V$. Given a collection of clusters $\mathcal{S}$, let $Diam(\mathcal{S}) = max_i(Diam(S_i))$ and $Rad(\mathcal{S}) = max_i(Rad(S_i))$. For every vertex $v \in V$, let $deg_{\mathcal{S}}(v)$ denote the degree of v in the hypergraph $(V, \mathcal{S})$, i.e., the number of occurrences of v in clusters $S \in \mathcal{S}$. The *maximum degree* of a cover $\mathcal{S}$ is defined as $\Delta(\mathcal{S}) = max_{v \in V}(deg_{\mathcal{S}}(v))$. Given two covers $\mathcal{S} = \{S_1, \ldots, S_m\}$ and $\mathcal{T} = \{T_1, \ldots, T_k\}$, we say that $\mathcal{T}$ *subsumes* $\mathcal{S}$ if for every $S_i \in \mathcal{S}$ there exists a $T_j \in \mathcal{T}$ such that $S_i \subseteq T_j$.

3.1.2 Hierarchical Directories

The hierarchical directory is based on the concept of a *m-regional covering.* A m-regional covering $\mathcal{T}$ is a covering with the following properties. Let $dist(v,u) \le m$. Then there exists a cluster $T \in \mathcal{T}$ such that $v \in T$ and $u \in T$. An m-regional covering is constructed using the following Theorem proved in [?].

Theorem 3.1 *Given a graph $G = (V,E)$, $|V| = n$, a cover $\mathcal{S}$ and any integer $b \ge 1$, it is possible to construct a cover $\mathcal{T}$ that satisfies the following properties:*

(1) $\mathcal{T}$ subsumes $\mathcal{S}$,

(2) $Rad(\mathcal{T}) \leq (2b-1)\, Rad(\mathcal{S})$*, and*

(3) $\Delta(\mathcal{T}) = O(b|\mathcal{S}|^{1/b})$*.*

An m-regional covering is constructed by letting $\mathcal{S} = \mathcal{N}_m(V)$ and applying Theorem 3.1. Based on the 2^i-regional covering $\mathcal{T}_i$, we define the regional directory $\mathcal{RD}_i$. Specifically, each cluster in $\mathcal{T}_i$ designates one of its members a the *cluster center*. Now, the regional directory is defined by the quantities $write_v[i]$, $read_v[i]$, and $synch_v[i]$, for each node v. In particular, the set $read_v[i]$ consists of the cluster centers of clusters $T \in \mathcal{T}_i$ such that $v \in T$, and $write_v[i]$ and $synch_v[i]$ are each the cluster centers of any cluster $T \in \mathcal{T}_i$ such that $\mathcal{N}_{2^{i+1}}(v) \subseteq T$. Intuitively, $\mathcal{RD}_i$ can be view as a directory where registrations to the directory are recorded at the cluster centers $write_v[i]$ and searches for registration are conduced by checking the cluster centers in $read_v[i]$. The construction of the clusters insures that a searching node will find any registration of a node that is within distance 2^i. The cluster center $synch_v[i]$ is used to synchronize concurrent search.

The following lemma bounds the number of $synch_v[i]$ cluster centers of any neighborhood.

Lemma 3.2 *Consider the neighborhood* $N_m(v)$*. Define* x *such that* $2^{x-1} \leq m \leq 2^x$*. Now* $H = \{T \mid T = synch_u[x] \text{ for any } u \in N_m(v)\}$*. Then* $|H| = O(b|n|^{1/b})$*.*

Proof: Consider any node $u \in N_m(v)$. Let $synch_u[x]$ be the cluster center of cluster T_u. Since $N_m(v) \subseteq T_u$ and $u \in N_m(v)$, we conclude that $v \in T_u$. The result now follows from part 3 of Theorem 3.1. ■

Finally, the hierarchical directory consist of the set of regional directories $\mathcal{RD}_i$ where $1 \leq i \leq \delta$.

3.1.3 Code Conventions

We use standard psuedocode with the following additional constructs. The construct **at** has the semantics of a remote procedure all. The indented lines following the **at** w: construct are executed at node w. The **find then else** construct is similar to the **if then else** construct. In particular, if the **find** command is successful, the **then** commands are executed, otherwise the **else** commands are executed. The **wait until** command, waits until the specified condition is satisfied. The waiting operation must be allowed to proceed before the condition has a chance to change in such a way that it is no longer satisfied.

There are several notational conventions. A variable that is subscripted by a node name, v, is a static variable that can be accessed by any operation executing at node v. Variables without subscripts are local to the operations. Finally, the atomic steps of our concurrent operations are specified as follows. Any sequence of commands is atomic until it reaches a **wait until** command where the condition is false or a command that requires the sending of a message.

3.2 Find

The `find` operation uses the hierarchical directory to locate a node that is currently registered in the hierarchical directory. The operation return the name of such a node. This section explains the code for the `find` operation given in Figure 1. The inherent cost of a `find` operation is the weighted distance to the closest registered node.

The operation proceeds by searching each level of the hierarchical directory, i.e. each regional directory $\mathcal{RD}_i$, in increasing order from 1 to δ. At level i the code for say node v checks the cluster centers in $read_v[i]$ for current registrations. Such a registration is found if the **find** command is successful at one of the cluster centers, i.e. at cluster center w, **find** returns a $P_w \in ptrset_w[i]$.

The code maintains the following invariant, If nodes v and u are concurrently executing `find` operations with $mode = insert$ and $synch_v[i] = synch_u[i]$, only one of the two nodes will search a regional directory $\mathcal{RD}_j$ where $j > i$. The invariant is ensured by the code block that succeeds the two phase search of $\mathcal{RD}_i$. Specifically node v checks with the cluster center $synch_v[i]$ to see if there are any concurrent searches pending. If so, $searching \neq nil$, and v is added to the set $synchset$. Otherwise $searching$ is set to v so that subsequent queries to $synch_v[i]$ will find $searching \neq nil$. If node v is added to the set $synchset$, node v does not search any of the higher regional directories. Rather, if $searching = w$, it waits until w, upon completing its insert operation, deletes v from the set $synchset$. In this case the `find` operation at v will return w.

3.3 Copy

The `copy` operation executed at node v inserts node v into the hierarchical directory. Once inserted node v can be returned as the result of a `find` operation. This section describes the code or the `copy` operation give in Figure 2. The inherent cost of a `copy` operation operation is the weighted distance to the the closest registered node.

The `copy` operation is best understood by first describing the concept of a coverset. A coverset is maintained for each regional directory. Consider the coverset for regional directory $\mathcal{RD}_i$. It consists of a forest with the following properties. If v is the root of a tree in the forest, then v is registered at the cluster center $write_v[i]$. Furthermore, the weighted depth of each tree is at most 2^{i+1}. If node v is in the coverset for $\mathcal{RD}_i$, its parent id given by $parent_v[i]$ and its children are contained in the set $ptrset_v[i]$. The actual elements of the set $ptrset_v[i]$ are structures with the fields $node$, which gives the name of the child, and $dist$ which gives the weighted distance between v and the child.

The `copy` operation at node v begins with a $\texttt{find}_v$ operation to locate a node from which the data copy will occur. Once such a node, say u, is found, the `addCoverSet` operation is executed. This operation attaches v to the coversets at each of the levels. `addCoverSet` attaches v to the coversets one level at a time. Consider the coverset for regional directory

$\mathcal{RD}_i$. Node v chooses the node, *node*, returned by the `find` operation as its parent in the coverset, sets $parent_v[i] = node$ and adds itself to the child set $ptrset[i]$ maintained at its parent. Next, v checks the weighted depth of the tree to which it has just attached. The depth of the tree is given by the variable $c_v[i]$. If the depth is greater than $2^{i+1}-2$ a `scanback` operation is initiated. The `scanback` operation walks the tree toward the root decrementing the $c_v[i]$ variable by 2^i at each node. Once the $c_v[i]$ variable at some node, say w, is less than or equal to 0, the subtree rooted at that node is detached from its current tree and registered at the cluster center $write_w[i]$.

Next, the `copy` operation contacts the cluster center $synch_v[i]$ in each $\mathcal{RD}_i$ up to level *level* to signal the completion of the `copy` operation to nodes that are waiting.

3.4 Correctness

Lemma 3.3 *In any concurrent execution α of* `fc`*, each* `find` *and* `copy` *operation terminates successfully.*

Proof: ∎

3.5 Complexity

We now consider the complexity of the algorithm.

Let α be a concurrent execution consisting only of `copy` operations. In this case α is said to be a `copy`-execution.

Lemma 3.4 *Let α be a* `copy`*-execution. Let $v \in V$ execute a* `copy` *operation in α and let the* `find` *operation executed by v's* `copy` *operation return the pair $(node_v, level_v)$. Then there are at most $O(b|n|^{1/b})$ nodes $u \in N_{2^{level_v}}(v)$ such that u executes a* `copy` *operation in α and the* `find` *operation executed by u's* `copy` *operation returns the pair $(node_u, level_u)$ such that $level_u \geq level_v$.*

Proof: This is a simple counting argument that makes use of Lemma 3.2 and the synchronization that the code does using the *synch* nodes. ∎

Lemma 3.5 *Let α be a* `copy`*-execution. Let the initial state of α, $first(\alpha)$, contain one data copy at node $v(\alpha)$. Then $cost(\alpha) = O((b|n|^{1/b})^2(2b-1))\,\mathcal{H}(n)\,opt(\alpha)$ for any b such that $1 \leq b \leq n$.*

Proof: Let $v \in V$ execute a `copy` operation in α and let the `find` operation executed by v's `copy` operation return the pair $(node_v, level_v)$. Let $x = level_v$. The cost associated with v's copy operation in α is the following:

$$\sum_{i=1}^{x}\left(4\,cost(v, synch_v[i]) + \sum_{w \in read_v[i]} (4\,cost(v,w))\right) + 2\,cost(v, node_v).$$

The first term covers the cost arising from the communication with the cluster centers in $read_v[i]$ and $synch_v[i]$ while the second term covers the cost of actually copying the data from node $node_v$ to node v. [[[Missing from the expression is the cost of updating the coversets and the additional cost arising from the fact that the amount of data copied from node $node_v$ to node v may be large. Both of these things are easy to add and to not change the order of the expression. I just have not developed the correct notation yet.]]] Using conditions 2 and 3 of Theorem 3.1 we can bound the expression in the outer sum by $O(b|n|^{1/b}(2b-1))2^i$. The second term is bounded by $O(2b-1)2^x$. The cost associated with the `copy` operation by v in α can now be expressed as follows:

$$\begin{aligned} cost(v) &= \sum_{i=1}^{x} O(b|n|^{1/b}(2b-1))2^i + O(2b-1)2^x \\ &= O(b|n|^{1/b}(2b-1)) \sum_{i=1}^{x} 2^i + O(2b-1)2^x \\ &= O(b|n|^{1/b}(2b-1))2^x. \end{aligned}$$

In order to establish our competitive result, we now must relate $cost(v)$ with the optimal cost for α, $opt(alpha)$. Let $S_v(\alpha)$ be the subset of nodes executing a `copy` operation in α with the following property. If $u \in S_v(\alpha)$ and the `find` operations executed by u's `copy` operation returned $(node_u, level_u)$, then $level_u \geq x$. First consider the case where $|S_v(\alpha)| - O(b|n|^{1/b}) \geq 0$. The case $|S_v(\alpha)| - O(b|n|^{1/b}) < 0$ is discussed later. By Lemma 3.4, $|S_v(\alpha) \cap N_{2^x}(v)| = O(b|n|^{1/b})$. This shows that there are at least $|S_v(\alpha)| - O(b|n|^{1/b})$ elements of S_v that are 2^{x-1} separated from v, i.e., there are $|S_v(\alpha)| - O(b|n|^{1/b})$ elements $u \in S_v(\alpha)$ such that $dist(u,v) \geq 2^x$. Now consider any node u that is 2^{x-1} separated from v and repeat the same argument. In this way we show that there are at least $\lfloor \frac{|S_v(\alpha)|}{O(b|n|^{1/b})} \rfloor$ elements of S_v that are 2^{x-1} separated.

Now let $S(\alpha)$ be the set of nodes that execute a `copy` operation in α. It is easy to see that $opt(\alpha) \geq MST_G(\{v(\alpha)\} \cup S(\alpha))$. Now we can conclude that:

$$2^{x-1} \lfloor \frac{|S_v(\alpha)|}{O(b|n|^{1/b})} \rfloor \leq MST_G(\{v(\alpha)\} \cup S(\alpha)) \leq opt(\alpha).$$

$$\Rightarrow 2^x \leq 2\, opt(\alpha) \lceil \frac{O(b|n|^{1/b})}{|S_v(\alpha)|} \rceil$$

Using this result we can now conclude that

$$\begin{aligned} cost(v) &= O(b|n|^{1/b}(2b-1))2^x \\ &= O(b|n|^{1/b}(2b-1))\, opt(\alpha) \lceil \frac{O(b|n|^{1/b})}{|S_v(\alpha)|} \rceil \end{aligned}$$

Note that the above equation only holds when $|S_v(\alpha)| - O(b|n|^{1/b}) \geq 0$. Now define $S'(\alpha) \subseteq S(\alpha)$ such that $u \in S'(\alpha)$ iff $|S_u(\alpha)| - O(b|n|^{1/b}) \geq 0$. Now the cost of the `copy` operations for all node in $S'(\alpha)$, $cost(S'(\alpha))$, is given by the following:

$$\begin{aligned} cost(S'(\alpha)) &= \sum_{v \in S'(\alpha)} O(b|n|^{1/b}(2b-1))\, opt(\alpha) \lceil \frac{O(b|n|^{1/b})}{|S_v(\alpha)|} \rceil \\ &= O((b|n|^{1/b})^2(2b-1))\, opt(\alpha) \sum_{v \in S'(\alpha)} \frac{1}{|S_v(\alpha)|} \\ &= O((b|n|^{1/b})^2(2b-1))\, \mathcal{H}(n)\, opt(\alpha). \end{aligned}$$

Finally, consider the case where $|S_v(\alpha)| - O(b|n|^{1/b}) < 0$. Let $S''(\alpha) = S(\alpha) - S'(\alpha)$. Let $u \in S''(\alpha)$ be the node such that $level_u \geq level_w$ for all $w \in S''(\alpha)$. Then it is easy to show

$$\begin{aligned} cost(S''(\alpha)) &= cost(u)|S''(\alpha)| \\ &= cost(u) O(b|n|^{1/b}) \\ &= O(b|n|^{1/b}(2b-1))\, opt(\alpha) O(b|n|^{1/b}) \\ &= O((b|n|^{1/b})^2(2b-1))\, opt(\alpha). \end{aligned}$$

Finally, we conclude that

$$cost(\alpha) = cost(S'(\alpha)) + cost(S''(\alpha)) = O((b|n|^{1/b})^2(2b-1))\, \mathcal{H}(n)\, opt(\alpha).$$

■

4 The Code

References

[AADW94] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. Proc. 35th IEEE Symp. on Found. of Comp. Science, 1994.

[AKP92] Baruch Awerbuch, Shay Kutten, and David Peleg. Online load balancing in a distributed network. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 571–580, 1992.

[AL94] Baruch Awerbuch and Tom Leighton. Improved approximation algorithms for the multicommodity flow problem and local competitive routing in dynamic networks. In *Proc. 26th ACM Symp. on Theory of Computing*, May 1994.

```
find_v()
    node := nil; i := 0
    repeat
        i := i + 1

        \* The i-level directory search, phase 1 *\
        forall w ∈ read_v[i]
            at w: find u ∈ ptrset_w then node := u
        until node ≠ nil

        \* Synchronize before moving to i + 1-level directory search *\
        if node = nil then
            w := synch_v[i]
            at w: if searching_w[i] ≠ nil
                        then node := searching_w[i]
                             synchset_w := synchset_w ∪ {v}
                             wait until v ∉ synchset_w
                        else if status_v = insertpending then searching_w := v

    until node ≠ nil
    return (node, i)
```

Figure 1: Code for $\texttt{find}_v$.

```
copy_v()
    status_v := insertpending
    (node, level) = find_v()
    copyData(node)
    addCoverSet(node)

    \* Inform waiting node of insert completion *\
    forall i := 1 to level
        w := synch_v[i]
        at w: searching_w = nil
            synchset_w := ∅
```

Figure 2: Code for $\texttt{copy}_v$.

[ALKK88] Anant Aggarwal, Beng-Long Lim, David Kranz, and John Kubiatowicz. Evaluation of directory schemes for cache coherence. In *Proceedings of 15th International Symposium on Computer Architecture*, New York, jun 1988. IEEE.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *J. of the ACM*, 32(4):804–823, October 1985.

[BBK+90] S. Ben-David, A. Borodin, R.M. Karp, G. Tardos, and A. Wigderson. On the power of randomization in online algorithms. In *Proc.of the 22nd Ann. ACM Symp. on Theory of Computing*, pages 379–386, may 1990.

[BFR92] Yair Bartal, Amos Fiat, and Yuval Rabani. Competitive algorithms for distributed data management. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 39–50, 1992.

[CFKA90] David Chaiken, Craig Fields, KiyoshiKurihara, and Anant Aggarwal. Directory-based cache-coherence in large-scale multiprocessors. In *IEEE Computer*, volume 23, pages 41–58, jun 1990. number 6.

[FKL+88] Amos Fiat, Richard Karp, Michael Luby, Lyle McGeoch, Daniel Sleator, and Neal E. Young. Competitive paging algorithms. unpublished, 1988.

[Gal82] Robert G. Gallager. Distributed minimum hop algorithms. Technical Report LIDS-P-1175, MIT, Lab. for Information and Decision Systems, January 1982.

[JLGS90] David James, Anthony Laundrie, Stein Gjessing, and Gurindar S. Sohi. Distributed-directory scheme: Scalable coherent interface. *IEEE Computer*, pages 74–77, jun 1990.

[KP94] Elias Koutsoupias and Christos Papadimitriou. On the k–server conjecture. In *Proc. 26th ACM Symp. on Theory of Computing*, May 1994.

[LEH85] K.A. Lantz, J.L. Edighoffer, and B.L. Histon. Towards a universal directory service. In *Proceedings of 4th PODC*, pages 261–271, Calgary, Alberta, Canada, August 1985.

[LLG+90] L. Lenoski, J. Laundo, K. Gharachorloo, A. Gupta, and J.Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proc. of 17th Intern. Symp. on Computer Architecture*, pages 148–159, 1990.

[MMS88] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms or on-line problems. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 322–333. ACM SIGACT, ACM, May 1988.

[PY93] Christos Papadimitriou and Mihalis Yannakakis. Linear programming without the matrix. In *Proc. 25th ACM Symp. on Theory of Computing*, May 1993.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. of the ACM*, 28(2):202–208, 1985.

[XP92] X.Deng and Christos Papadimitriou. On the value of information. In *Proceedings of the 12th IFIPS Congress, Madrid. Also, in Proceedings of the World Economic Congress, Moscow, 1992.*, 1992.