Linear Broadcasting and $N \log \log N$ Election in Unoriented Hypercubes *

Stefan Dobrev, Peter Ružička

Institute of Informatics, Faculty of Mathematics and Physics, Comenius University, Mlynská dolina, Bratislava, Slovak Republic

Abstract. In this paper, we provide efficient broadcasting and election algorithms in unoriented hypercubes. First, $O(N)$ broadcasting and traversing algorithms are introduced, where N is the number of hypercube vertices. This answers affirmatively the open question stated in [Tel95a] whether linear-message broadcasting and traversing can be achieved on hypercubes without sense of direction. Moreover, by exploiting new techniques we designed an $O(N \log \log N)$ leader election algorithm. This is the first known solution being able to exploit graph-theoretic properties of unoriented hypercubes such that it outperforms algorithms designed for general graphs [GHS83].

1 Introduction

The impact of the structural information knowledge upon the communication complexity of distributed algorithms, started by [San84], has been intensively studied in recent years. Many results, showing whether some given type of structural information about the network topology helps (or does not help) to decrease the communication complexity, have been obtained for a particular problem, a particular class of network topologies and a particular structural information.

For example, in a *complete graph*, a leader can be elected using $O(N)$ messages if sense of direction is available [LMW86], while $\Omega(N \log N)$ messages are required otherwise [KMZ84]. On *general graphs* with sense of direction a $O(N \log N)$ solution has been given in [MS94], while $\Omega(e + N \log N)$ messages are needed without sense of direction [GHS83]. In *torus*, leader election can be done in $O(N)$ messages without sense of direction [Pet85, Mans96]. For more details on this topic see [Tel95a].

In [FM96], [Tel95b], authors presented linear-message leader election algorithm in hypercubes with sense of direction. Broadcasting, traversing and even constructing a Hamiltonian circuit in oriented hypercubes can be easily accomplished in $O(N)$ messages. The question whether sense of direction indeed helps in hypercubes remained open, as there is lack of efficient algorithms for unoriented case. The only known results were obtained as direct applications of

* The research was partially supported by EU Grant No. INCO-COP 96-0195 "ALTEC-KIT" and by Slovak VEGA project 1/4315/97.

general algorithms, e.g. [GHS83] on unoriented graphs, which in this case lead to unsatisfactory $O(N \log N)$ even for broadcasting and traversing.

In this paper we give an answer to certain questions stated in [Tel95a]. In Section 2 we give an $O(N)$ algorithm for broadcasting and traversing in unoriented anonymous hypercubes, showing that sense of direction is not necessary to solve these problems in asymptotically optimal communication[2]. Moreover, $O(N \log\log N)$ leader election algorithm in unoriented hypercubes is presented in Section 3, thus asymptotically narrowing a gap between oriented and unoriented case. These algorithms exploit the structure of the hypercube in a novel way, bypassing the use of its obvious dimensional structure which is not available in unoriented case.

1.1 The Hypercube and its Graph-Theoretic Properties

An n-dimensional *hypercube* is an undirected graph Q_n consisting of $N = 2^n$ vertices and $\frac{1}{2}Nn$ edges. Vertices can be addressed from 0 to $2^n - 1$ such that there is an edge between any two vertices iff the binary representation of their addresses differs by exactly one bit. Q_n is regular of degree n and it has diameter n.

We start with some graph-theoretic results about hypercubes which are used in Section 2. Their proofs are given in the full version of the paper [DR97].

A set $\mathcal{D}$ of vertices is called a *dominating set* of Q_n iff for each vertex v from Q_n it holds $v \in \mathcal{D}$ or there is $u \in \mathcal{D}$ such that (u, v) is an edge in Q_n.

Lemma 1. *There is a dominating set $\mathcal{D}$ of n-dimensional hypercube Q_n of the size at most $\frac{4N}{n+1}$.*

Next we give a 1-level clustering for hypercubes which is useful in the design of efficient broadcasting algorithm in Section 2.1.

Lemma 2. *There exists a division of the n-dimensional hypercube Q_n into clusters such that (1) each cluster has radius at most 8, (2) there are $O(N/n^4)$ clusters.*

1.2 Network Model

The computational model is a hypercube network [Tel94b]. That means interconnection topology of the network is a hypercube graph Q_n. Each processor has unique identification number from some totally ordered set ID. Identities are without topological significance. A processor knows only its own identity, and in particular it does not know identities of its neighbours. Each processor can distinguish its links by uninterpreted labels, which have no topological meaning. Every processor performs the same algorithm. In the *broadcasting* and *traversing* problems, there is one distinguished processor – *initiator* that starts

[2] The same result was independently obtained by Diks, Kranakis and Pelc [DKP96].

a computation. All other processors are sleeping and they join the computation after receiving the first message. In the *leader election* problem, any non-empty subset of processors may initiate computation.

In the *broadcasting* problem we require that each processor receives some message from initiator. To solve the *traversing* problem there should be some center of activity that starts at initiator and sequentially traverses the whole network. Solving *leader election* problem means that starting from the initial configuration where all processors are in the same state, the system should reach a configuration where exactly one processor is in the *leader* state and all other processors are in the *defeated* state.

The measure of *communication complexity* is the maximum number of messages sent by algorithm over all possible executions. Messages are supposed to be of length $O(n)$.

1.3 Notions and Terminology Used

For $i \leq n$, by $\mathbf{e}_i$ we denote binary n-vector with a 1 in the i-th position and 0s elsewhere and by e_i corresponding binary string. Without loss of generality we assume that at each vertex local link labels are from $\{e_1, \ldots, e_n\}$, but there is no correlation between local link labels at different vertices.

By a *world according to* x we mean an assignment of vertex addresses from 0 to $2^n - 1$ to all vertices of Q_n and an assignment of link labels from e_1 to e_n to all links such that x is addressed by 0, links adjacent to x are labeled by initial local labels, and assignments of vertex addresses and link labels are globally consistent (i.e. addresses of two neighbouring vertices differ in exactly one bit position determined by the label of the link that connects them).

Fixed world according to x can be described by a pair (σ_x, β_x), where $\sigma_x(v)$ denotes a permutation that maps local link labels at v to link labels in the world according to x and $\beta_x(y)$ denotes a vertex with address y in the world according to x. So by $\beta_x^{-1}(y)$ we denote an address of the vertex y in the world according to x. For a link $h = (u, v)$ by h_u we denote a local label of h in u and by $\sigma_x(u)[h_u]$ a label of link h in the world according to x.

By $Send(h, message)$ we mean a procedure sending a message $message$ along the link h.

2 Broadcasting and Traversing

Throughout this section we consider an n-dimensional unoriented hypercube with a fixed vertex called *initiator*.

2.1 Broadcasting in Unoriented Hypercube

The algorithm for efficient broadcasting in an unoriented hypercube is based on the efficient construction of the optimal routing path from the initiator to the destination. It is performed by the procedure $Route()$ and the details are presented in the following lemma.

Lemma 3. *Consider an unoriented Q_n with an initiator x. Let y_x be an address of a vertex y in Q_n in a world according to x. Then there exists an algorithm which computes an optimal routing path from the initiator x to the destination y using $O(nd^2)$ messages, where d is the distance between x and y in Q_n.*

Proof: We can restrict to the case $y_x = 1^d 0^{n-d}$. If y_x is not in this form, we can permute link labels of x (in a world according to x) to achieve the appropriate address form [Lei92]. The main idea of the proof is to explore only the small part of the hypercube. We will successively find vertices with addresses from $\{0^i 1^j 0^{n-i-j} \mid i = 1, \ldots, d, j = 0, \ldots, d-i\}$. The vertex v at the distance i from the initiator x can compute its address from the known addresses of its two neighbours which are at the distance $i-1$ from the initiator x. The address of v is obtained as the binary **or** of addresses of both of these neighbours.

Algorithm *Route*(y_x);
Input: Invoked at the initiator x with an address 0^n. An address $y_x = 1^d 0^{n-d}$ of a destination vertex y.
Output: An implicit routing path from x to y.
Method:

At the initiator x:
{*Send link's labels (neighbour's addresses) to the first d neighbours.*}
Name$_x$:= 0^n;
for $i := 1$ **to** d **pardo** Send(e_i,(start,e_i,d));

At a vertex v:

(1) **Upon receiving a message (start, e_i, d) via the link h:**
{*v is a neighbour of the initiator. Inform all neighbours of v about its address.*}
Name$_v$:= e_i;
Label$_{v,h}$:= e_i;
for $j := 1$ **to** n **pardo**
  if $i \neq j$ **then** Send(e_j,(Name$_v$,d));

(2) **Upon receiving a message (x, d):**
{*For vertices that did not yet computed their names.*}
Remember this message.
Proceed if you had received two messages of the form $0^i 1^j 0^{n-i-j}$ (received via the link h_1) and $0^{i+1} 1^j 0^{n-i-j-1}$ (received via the link h_2) where $0 \leq i \leq d-j$.
Name$_v$:= $0^i 1^{j+1} 0^{n-i-j-1}$;
Label$_{v,h_1}$:= e_{i+j+1};
Label$_{v,h_2}$:= e_{i+1};
if $j+1 = d$ **then** Send(h_1,(Name$_v$,d)), Send(h_2,(Name$_v$,d)) **halt**
else
  for $k := 1$ **to** n **pardo** Send(e_k, (Name$_v$, d));

(3) **Upon receiving a message (x, d) via the link h:**
{*For vertices that had already computed their names.*}
Label$_{v,h}$:= x **xor** Name$_v$;

Correctness: Let $V_d = \{0^i 1^j 0^{n-i-j} \mid i = 1,...,d, j = 0,...,d-i\} \cup \{0^n\}$ and $E_d = \{(x,y) \mid x,y \in V_d\}$. To prove the correctness of the *Route*() it is enough to show (1) $\forall v \in V_d : \text{Name}_v = \beta_x^{-1}(v)$, (2) $\forall h \in E_d, h = (u,v) : \text{Label}_{u,h_u} = \sigma_x(u)[h_u]$. We prove (1) by contradiction. Let w be the first vertex for which it holds $\text{Name}_w \neq \beta_x^{-1}(w)$ in *Route*(). Its name is computed as a boolean **or** of names of some neighbouring vertices u and v. By assumption, u and v correctly computed their names. Let $\text{Name}_u = 0^i 1^j 0^{n-i-j}$ and $\text{Name}_v = 0^{i+1} 1^j 0^{n-i-j-1}$. There are just two vertices that can receive direct messages from both u and v. These vertices have addresses Name_u **and** $\text{Name}_v = 0^{i+1} 1^{j-1} 0^{n-i-j}$ and Name_u **or** $\text{Name}_v = 0^i 1^{j+1} 0^{n-i-j-1}$. As w wrongly computed its name, it should have the address $w' = \text{Name}_u$ **and** $\text{Name}_v = 0^{i+1} 1^{j-1} 0^{n-i-j}$. But according to the algorithm, u will compute its name after receiving messages from two vertices, one of which has the name w'. (The same holds for v.). This means that the vertex w' had already computed its name and cannot be w. Thus, for w there remains the only possibility to have the address Name_u **or** Name_v. But in this case w correctly computes its name, a contradiction. Further, (2) follows directly from (1) and the description of the algorithm.

Communication complexity: Every vertex from V_d sends n messages, initiator x sends d messages and other vertices send no messages. Thus, the total communication complexity is $d + n|V_d| \in O(nd^2)$, because $|V_d| \in O(d^2)$.

Lemma 3 allows us to call subroutine $FarSend(destination, message)$. This subroutine first calls algorithm *Route*(*destination*) of Lemma 3 and then sends *message* via the optimal route to its *destination*. Note, that if we need to send a message via this optimal path once more it is sufficient to use only the second part of $FarSend()$ which we refer to as $FarJump()$. We just need to keep in mind that the cost of $FarSend()$ is not d, but $O(nd^2)$ messages. To avoid the confusion caused by multiple and concurrent calls of $FarSend()$, we mark internal messages of each such a call by the name of sending vertex v and the counter c_v which is incremented in each call of $FarSend()$ in v.

Basic Non-Optimal Broadcasting Vertices in $\mathcal{D}_{Q_n}$ are called *important*. The main idea of the basic algorithm is the following. The initiator sends a message *expand* using subroutine $FarSend()$ to all *important* vertices. When a vertex receives *expand* message, it sends a message *hello* to all its neighbours. Because of the covering property of $\mathcal{D}_{Q_n}$, all vertices are visited.

Unfortunately, basic algorithm cannot be directly implemented in an efficient way. The term $O(nd^2)$ in Lemma 3 causes that the total communication complexity increases to $O(Nn^2)$, what is far from the optimality.

Brief Description of the Optimal Broadcasting Algorithm The source of inefficiency in the basic approach was the term $O(nd^2)$. To overcome this difficulty, *expand* messages are sent only to nearby vertices, within a constant distance. This can be solved by 1-level clustering method. First, appropriate messages are sent to centers of clusters and then these centers forward *expand* messages to nearby important vertices.

It is necessary to achieve globally consistent orientation of cluster centers to allow consistent forwarding to important vertices. This can be achieved by procedure $FarOrient(destination)$, which assigns to the destination vertex its address in the world according to invocation vertex and sets its link labeling consistent with the link labeling of invocation vertex.

2.1.0.1. FarOrient. $FarOrient()$ can be implemented as follows: Send an address not only to the destination vertex, but also to all its neighbours. Then destination vertex asks all its neigbours about their addresses and from received replies and its own address it computes labels of incident links. Such an implementation of $FarOrient()$ requires $n+1$ calls of $FarSend()$ and additional $2n$ messages for local lookup to compute the labels. Thus $FarOrient()$ can be implemented using $O((nd)^2)$ messages.

2.1.0.2. Broadcasting. The overall brief description of broadcasting algorithm:

1. Following Lemma 2 logically divide Q_n into $O(N/n^4)$ non-overlapping clusters, each of radius at most 8. By $\mathcal{CC}$ denote the set of addresses of all cluster centers.
2. At the initiator:
   For all $x \in \mathcal{CC}$: FarOrient(x); FarSend(x,center)
3. At a center v of a cluster $\{$*After delivering message (v,*center*)*$\}$*:*
   For all important vertices u in the cluster with center v: FarSend(u,expand)
4. At an important vertex w: $\{$*After delivering message (w,expand)*$\}$*:*
   Send *hello* to all neighbours.

Communication Complexity of the Broadcasting Algorithm There are three distinct phases of the algorithm:

1. Informing centers of clusters.
   The cost of each call of $FarOrient()$ is bounded by $O(n^4)$. There are at most $O(N/n^4)$ clusters, so the total communication complexity of this phase is within $O(N)$.
2. Informing important vertices.
   Each important vertex is within the constant distance from the center of its cluster, thus the corresponding $FarSend()$ needs just $O(n)$ messages. Due to Lemma 1 there are $O(N/n)$ important vertices, so the total communication complexity is again $O(N)$.
3. Final expansion of important vertices.
   Each important vertex sends at most n *hello* messages, there are $O(N/n)$ important vertices, that makes together again $O(N)$ messages.

Counting messages used in all three phases, we end up with just $O(N)$ messages.

2.2 Traversing in Unoriented Hypercubes

The broadcasting algorithm can be easily modified to solve the problem of traversing. Moreover, this modification does not increase asymptotical communication complexity. Description of this modification is given in the full version of the paper [DR97].

3 Leader Election

The leader election algorithm is a combination of two algorithms – *Warrior* and *King*. These two algorithms are combined following the well-known principle of accelerated cascading.

At the beginning *Warriors* are born at each vertex that starts a computation of leader election algorithm (that can be potentially all vertices in Q_n). Initially, the domain of the warrior consists of a singleton vertex at which the warrior has born. Each warrior tries to enlarge his domain to became sphere of radius 5. When he succeeds, he is promoted to a *King* and further executes the algorithm for kings.

The goal of the *warrior phase* is to eliminate warriors in an efficient way such that only a small number of them became kings. In order to keep the communication complexity of this phase low, warriors build their domains compact. Domain of the warrior is always a sphere (with some exceptions on the border) with the center at the vertex where the warrior has born.

In the *king phase* every kingdom tries to expand until it rules the whole universe Q_n. The main difference from the warrior phase is that when two kingdoms meet, the whole defeated kingdom is appended to the winning kingdom. The point is that although this joining procedure is rather costly, the warrior phase assures that kingdoms are big enough and there are not so many of them, thus the total cost of joining procedure will not be too high.

3.1 The Warrior Phase

Warriors have the name *Warrior(size,v)*, where an attribute *size* is the size (i.e., the number of vertices) of the domain of this warrior and an attribute v is the identity of the birthplace of the warrior. By $\mathcal{D}_v$ we denote the domain of a warrior born at v. In a duel of two warriors, the one with larger domain is stronger, and if the domains are equal, birthplace identities are considered; moreover kings are stronger than warriors. This leads to the fact that the resulting leader is not necessary the one with the maximal initial *id*, but the one whose domain grows fastest.

We present an informal description of the algorithm for *warrior* born at v:

1. Mark v by your name.
2. Go to the last explored vertex and using sequential BFS find a vertex x not belonging to $\mathcal{D}_v$. (When expanding a vertex, examine unexplored links one–by–one, not in parallel.)

3. If x is marked by a stronger warrior then die; otherwise follow the links leading to the birthplace w of the former owner of x.
4. If w is marked by a stronger warrior then die; otherwise mark w by your name. (If the warrior from w is still alive, it will die after returning to w.)
5. Return to the vertex x, mark it by your name and label the link leading from x to $\mathcal{D}_v$. This ensures the existence of a tree rooted at v spanning $\mathcal{D}_v$.
6. Return to v. If v is marked by a stronger warrior then die; otherwise increase your *size* attribute by one.
7. If your domain is not a sphere of radius 5 then *go to 1.*
8. Start the algorithm for kings.

Lemma 4. *Using the warrior algorithm, the cost of building a domain of size k and radius d is $O(dk)$ messages.*

Proof: To every new vertex added to the domain we will charge the communication cost performed during the iteration of the loop in which this vertex was added to the domain, with exception of step 2. Because all other steps are clearly in $O(d)$, all we need is to show how to distribute potentially higher cost of step 2 among other vertices such that every vertex is charged by cost $O(d)$.

The cost of step 2 can be divided into two parts:

– exploring links at border vertices:
The cost of every trial to find a new neighbour will be charged not to the border vertex, but to the neighbour that was reached by this trial. Because the radius of the domain is d (and we use BFS), no vertex is charged here for more than d.
– traversing to another border vertex (when the previous border vertex already explored all its neighbours):
Traversal will be charged to the vertex that was fully explored. Because every vertex is explored just once, and the path from one such vertex to another one is bounded by $2d$, every vertex is charged here by additional $O(d)$.

Lemma 5. *Let each warrior promoted to king has the domain bounded by size K and radius by d. Assume d is $O(1)$. Then the total communication complexity of the warrior phase is $O(N \log\log N)$.*

Proof: The proof is based on the previous lemma. Let s_i be the number of domains that reached the size i. It holds $s_i \leq N/i$. The overall communication complexity of the warrior phase can be bounded by $\sum_{i=1}^{K} c \cdot i \cdot (s_i - s_{i+1})$, where c is the upper bound constant obtained in Lemma 4 and s_{K+1} be defined as 0. Then $\sum_{i=1}^{K} c \cdot i \cdot (s_i - s_{i+1}) = c \sum_{i=1}^{K} s_i \leq c \sum_{i=1}^{K} N/i = c \cdot N \sum_{i=1}^{K} 1/i \in O(N \log K)$. Because $K < n^d$ and $d \in O(1)$ we get $O(N \log K) = O(N \log n) = O(N \log\log N)$.

3.2 The King Phase

The goal of the king phase is to repeatedly merge kingdoms, until there remains only one kingdom, whose capital becames the leader. The computation is finished by broadcasting the identity of the leader.

During the whole king phase, the following *invariant* holds for each active kingdom $\mathcal{K}$:

1. The kingdom $\mathcal{K}$ is the union of spheres $S_i^{\mathcal{K}}$, each of the radius at most 5. (These spheres may not have complete outermost layers.) Let c_i be the center of the sphere $S_i^{\mathcal{K}}$ and c_0 denote the capital of $\mathcal{K}$. *Map*, stored in c_0, is a set $\{(\beta_{c_0}^{-1}(c_i), |S_i^{\mathcal{K}}|) \mid i \geq 1\}$.
2. There is a tree $\mathcal{T}_i^{\mathcal{K}}$ spanning $S_i^{\mathcal{K}}$ and rooted towards c_i, which is maintained at vertices of $S_i^{\mathcal{K}}$ by variables *Parent*.
3. There is a spanning tree $\mathcal{TC}^{\mathcal{K}}$ over all c_i, rooted towards c_0. (Links of $\mathcal{TC}^{\mathcal{K}}$ correspond to paths between centers, they are created using $FarSend()$ and can be optimally traversed using $FarJump()$).
4. Trees $\mathcal{T}_i^{\mathcal{K}}$ and $\mathcal{TC}^{\mathcal{K}}$ form a directed graph, denoted as $\mathcal{ST}^{\mathcal{K}}$, that spans the whole kingdom $\mathcal{K}$. (The depth of $\mathcal{TC}^{\mathcal{K}}$ is bounded by n, so the depth of $\mathcal{ST}^{\mathcal{K}}$ is bounded by $O(n^2)$ since each link of $\mathcal{TC}^{\mathcal{K}}$ corresponds to the path that is not longer than n and $\mathcal{T}_i^{\mathcal{K}}$, for all i, has the depth at most 5).

For brevity, centers of colonies and capitals (active and defeated) are called *essential* vertices.

Brief Description of the King Algorithm The algorithm is described in the token-driven framework. There is one token reserved for each kingdom. When two kingdoms meet, after performing the duel only one token survives.

We present the algorithm from the point of view of activities performed by the token, instead of the common view of computation executed at each vertex.

1. Mark your basic domain (inherited from the warrior phase) using sequential BFS and build its BFS spanning tree. If you find a mark of another kingdom, execute $Join()$ procedure.
2. Find a place for a new colony. Choose an address x such that the distance from your kingdom is at least 5. (This can be done by a local computation at the capital using Map.)
3. If you cannot find such an address, you are the only surviving king. Broadcast the identity of your capital. Halt.
4. Otherwise establish a colony.
   - Set up a center of the colony using $FarOrient(x)$.
   - Expand this colony using sequential BFS (and build its spanning tree) until either it becames a sphere of radius 4 or a mark of another *king* has been found.
5. In case of finding the mark of another *king*, execute $Join()$ procedure.
6. Go to 2.

In the $Join()$ procedure the weaker kingdom surrenders to the stronger one and its token is discarded. Map and a spanning tree of winning kingdom are updated. The kingdom is weaker if its size is smaller or if the sizes are equal and the identity of its capital is smaller.

Detailed Description of the King Algorithm

3.2.0.1. Variables used at a vertex of the hypercube Q_n. The following variables are used at each vertex:

- $Owner$ - the identification number of the owner of this vertex, inherited from the warrior phase. The owner can be a $king$ or a $warrior$.
- $Parent$ - a link of distributively stored rooted tree of a sphere.
- $TowardCollision$ - used for distributively stored path from the capital towards the place of collision.
- Map - at an active capital: an information about the topology of the kingdom in the form of a set of pairs $(center, sphere - size)$.
- $AttackedBy$ - at an active capital: a list of arrived kings waiting for the return of home king to perform a duel.
- and others - used by BFS, $Warrior()$, $FarSend()$, $FarJump()$, $FarOrient()$ and $Broadcast()$.

3.2.0.2. Attributes of a token. Each active kingdom has its token (called $king$) that travels through the graph and tries to expand its kingdom. When two tokens are handled simultaneously, the second one is named $attacker$. The token has the following attributes:

1. $status$ - what is $king$ currently doing.
2. id - identity of the capital of the kingdom.
3. $size$ - the size of the kingdom, i.e. the sum of sizes of all spheres listed in Map.
4. $added$ - the number of recently added vertices not yet accounted at Map.
5. Map - a set giving an information about the defeated capital to winner.
6. and other attributes used during BFS of base kingdom and colonies.

3.2.0.3. Building a base kingdom. When the $warrior$ is promoted to the $king$, his base kingdom consists of a single vertex - the capital c_0. The $king$ builds his base kingdom using BFS to depth 5. During his traversal $king$ marks new visited vertices of the base kingdom by setting vertex's $Owner$ to $king.id$ and constructs the BFS spanning tree by setting vertex's $Parent$ to the last traversed link. The number of visited vertices is stored in $king.added$. After accomplishing successful traversal of the whole sphere of the depth 5, $king$ returns to its capital using spanning tree of its kingdom and the pair $(0, king.added)$ is added to Map.

3.2.0.4. Building a colony. By a computation at the capital (using Map) an address x of a center of a colony is choosed such that its distance from the kingdom is at least 5. If such an address cannot be chosen, you are the only surviving $king$. (See that there could be a place for a new colony (of the radius 4) not computable from the Map due to uncomplete outermost layers of spheres of the kingdom, but certainly there is not enough place for a new kingdom (of the radius 5)). In this case, broadcast the identity of the capital (the leader) of the kingdom (it reaches also vertices which are not a part of the kingdom) and halt. The colony is built using BFS from the center to the depth 4 in a similar way as the base kingdom from the capital. If the colony has been successfully built, the $king$ returns to its capital, the pair $(x, king.added)$ is added to Map and the $king$ proceeds with building another colony.

3.2.0.5. Resolving collisions. If the $king$ finds a mark of another kingdom (by testing ($Parent$ is set) **and** ($Owner \neq king.id$)) during the building of base kingdom or colony, joining of these two kingdoms is performed.

1. Return to your capital using spanning tree of your kingdom and mark the path towards the place of collision.
2. Update Map by adding a pair $(c, king.added)$, where c is the center of currently built sphere.
3. Resolve $Duel$ if there are kings waiting for you. ($AttackedBy \neq \emptyset$)
4. First follow marked links towards the place of collision. Then follow the spanning tree of attacked kingdom until you arrive to its capital. Before traversing a link, cast FarOrient() on vertex one step ahead, so that you always know where you are going and what is the relationship between the orientation at current vertex and the orientation at your capital.
5. Resolve $Duel$ at the capital of attacked kingdom.

3.2.0.6. Resolving duel. $Duel()$ is resolved whenever attacking $king$ arrives to the enemy capital. The goal of the duel is to decide who will win and to update Map and the spanning tree of winning kingdom.

If attacked $king$ is already at home, duel can be immediately performed by comparing $attacker$ and $king$. If $attacker$ is stronger, the token of $king$ is discarded, $attacker$ returns home using FarSend() (he already knows relative position of its capital), updates the spanning tree of the winning kingdom (by adding the spanning tree of defeated kingdom and a (logical) edge from defeated to winner capital) and also brings Map of defated kingdom, so Map of his kingdom can be updated (using the relationship between orientations of winner and defeated kingdoms). If $attacker$ is weaker, he returns home. At home he checks whether there is a stronger king waiting for the duel. (By testing $AttackedBy$, a list of kings waiting for duel.) If that is not the case, he surrenders to unsuccessfully attacked $king$ (by sending him Map of his kingdom and adding a (logical) edge from his capital to winner's capital into spanning tree of winner's kingdom. After receipt, this surrender token is processed and discarded.) If the stronger *king* waits at home, the *attacker* forgets about his unsuccessful attack and surrenders to this king.

Attacked *king* is not usually at home. In such a case *attacker* compares its size with the last known size of attacked kingdom using his *Map*. If *attacker* is weaker, he proceeds as in the previous case, otherwise he waits for attacked *king* (by adding into *AttackedBy*). It may happen that the *king* of attacked kingdom enlarged his kingdom during his expedition, so the rule "bigger wins" would not be ensured if the *attacker* just leaves the mark Die! and continues his own computation.

Correctness First we prove that the invariant holds during the computation of the algorithm *King*.

1. Each part that is added to the kingdom (i.e., when it is added to its *Map*) is either a sphere or a set of spheres, so the kingdom is always a set of spheres.
2. These spanning trees are built during building base kingdom and colonies.
3. The edges of this spanning tree are created either when colony is established or defeated kingdom is connected to the winner.
4. It is sufficient to show that the (logical) depth of $\mathcal{TC}^{\mathcal{K}}$ is bounded by n. All inner edges of $\mathcal{TC}^{\mathcal{K}}$ are directed from defeated capital to the winner capital. Since larger kingdom always wins, parent vertex corresponds to the kingdom which is at least twice as big as the defeated one. There are 2^n vertices, so the depth is bounded by n.

We need to show that *King* algorithm always terminates and there is exactly one vertex selected as the leader.

Termination: It is sufficient to show that deadlock cannot occur. Since the larger kingdom always waits for the smaller one (sizes of Map-s are compared) and the *attacker* always carries actual size that is stored at its capital (that is the explanation for back-up return home after collision), deadlock is indeed avoided.

It is easy to see that there is always at least one active kingdom. A kingdom dies only when it collides with another kingdom. In such a case, one kingdom (the larger one) always survives.

Next, we need to show that if there are at least two kingdoms, they always collide. This follows from the fact that kingdoms grow and they have to be disjoined.

This also leads to the conclusion that there will be eventually a kingdom that cannot find a place for another colony. Since there is no place for another colony, there is no place for another kingdom, so this kingdom is a single surviving kingdom and it can declare its capital to be a leader.

Communication Complexity of the King Phase The total communication can be accounted to some of these tasks: (1) Building base kingdoms. (2) Establishing colonies that successfully grew to the full size. (3) Establishing colonies that did not grew to the full size. (4) Joining the kingdoms. (5) Broadcasting phase at the end.

1. The cost of building the base kingdom is clearly $O(n^5)$ - the size of an initial kingdom. Since there are $O(N/n^5)$ kingdoms (kingdoms do not overlap), the total cost is $O(N)$.
2. There are at most $O(N/n^4)$ such colonies, because the colonies do not overlap. The cost of building one such colony is $O(n^4)$ for the initial establishing and orientation (complexity of $FarOrient()$) and $O(n^4)$ for actual growth. So the overall communication complexity of this computation is $O(N)$.
3. The cost of establishing and growth of each such a colony is again $O(n^4)$, the difference is in the argument about their number: each colony that does not grew to the full size caused collision and death of one kingdom. (In fact there may be two colonies responsible for one death, because we did not consider attacking when attacked. This can also be handled by the algorithm, but it would just complicate it, without improving its asymptotical performance.) The number of such colonies is thus bounded by the initial number of kingdoms $O(N/n^5)$, thus the overall cost of this computation is $O(N/n)$.
4. The cost of each joining is bounded by $O(n^4)$, there are at most $O(N/n^5)$ joins, so the overall cost of this case is $O(N/n)$.
5. The broadcasting algorithm requires $O(N)$ messages.

Counting together we get that the total communication complexity of the king phase is $O(N)$ messages.

3.2.0.1. Bit communication complexity of the king phase. All messages are of the size $O(n)$, except when Map is beeing sent. Information about each sphere listed in Map is sent over at most $O(n^2)$ links (the depth of $\mathcal{ST}^{\mathcal{K}}$), there are at most $O(N/n^4)$ different spheres (either colonies or base kingdoms), so we can bound total bit complexity of sending Map-s by $O(n^2) \cdot O(n) \cdot O(N/n^4) = O(N/n)$. Clearly this does not influence the overall bit complexity of the king phase, which is $O(Nn)$.

Detailed algorithms for the King phase are given in the full version of the paper [DR97].

4 Conclusions

We have seen that there are efficient algorithms on hypercubes without sense of direction. We improved the performance of leader election to $O(N \log\log N)$ and also presented a linear-message broadcasting and traversing in this model, thus giving an answer to open questions formulated in [Tel95a].

Interesting unresolved problems in unoriented hypercubes are whether there is

- a linear-message election algorithm, and
- an $o(N \log N)$ algorithm for constructing a Hamiltonian circuit.

References

[DKP96] Diks, K. – Kranakis, E. – Pelc, A.: *Broadcasting in Unlabeled Networks.* Départment d'Informatique, Université du Québec à Hull, Technical Report, RR 96/12-5, December 1996.

[DR97] Dobrev, S. – Ružička, P.: *Linear broadcasting and* $N \log\log N$ *election in unoriented hypercubes.* Technical Report, Department of Computer Science, Comenius University, Bratislava, April 1997, p. 19.

[FM96] Flocchini, P. – Mans, B.: *Optimal elections in labeled hypercubes.* Journal of Parallel and Distributed Computing 33, 1996, pp. 76 – 83.

[GHS83] Gallager, R. G. – Humblet, P. A. – Spira, P. M.: *A distributed algorithm for minimum–weight spanning trees.* ACM Transactions on Programming Languages and Systems 5, 1983, pp. 66 – 77.

[KMZ84] Korach, E. – Moran, S. – Zaks, S.: *Tight lower and upper bounds for a class of distributed algorithms for a complete network of processors.* Symposium on Principles of Distributed Computing, 1984, pp. 199 – 207.

[Lei92] Leighton, F.T.: *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, 1992.

[LMW86] Loui, M.C. – Matsushita, T.A. – West, D.B.: *Election in complete networks with a sense of direction.* Information Processing Letters 22, 1986, 185 – 187. Addendum: Information Processing Letters 28, 1988, p. 327.

[MS94] Mans, B. – Santoro, N.: *On the impact of sense of direction in arbitrary networks.* Proc. of 14th International Conference on Distributed Computing Systems (ICDCS'94), Poznan, 1994, pp. 258 – 265.

[Mans96] Mans, B.: *Optimal Distributed Algorithms in Unlabelled Tori and Chordal Rings.* Proc. of the 3rd International Colloquium on Structural Information and Communication Complexity (SIROCCO'96), Carleton Press, Siena, Italy, June 1996, pp. 17-31.

[Pet85] Peterson, G. L.: *Efficient algorithms for elections in meshes and complete networks.* Technical Report TR140, Department of Computer Science, University of Rochester, Rochester, NY 14627, 1985.

[San84] Santoro, N.: *Sense of direction, topological awareness and communication complexity.* SIGACT NEWS 1, 1984, pp. 50 – 56.

[Tel94a] Tel, G.: *Network Orientation.* International Journal on Foundations of Computer Science 5, 1994, pp. 23 – 57.

[Tel94b] Tel, G.: *Introduction to Distributed Algorithms.* Cambridge University Press, Cambridge, 1994.

[Tel95a] Tel, G.: *Sense of Direction in Processor Networks.* In: SOFSEM'95, Theory and Practise of Informatics, LNCS 1012, Springer–Verlag, 1995, pp. 50 – 82.

[Tel95b] Tel, G.: *Linear election in oriented hypercubes.* Parallel Processing Letters 5, 1995, pp. 357-366.

This article was processed using the LaTeX macro package with SIROCCO style