Correcting Sorted Sequences in a Single Hop Radio Network*

Marcin Kik
Marcin.Kik@pwr.wroc.pl

Institute of Mathematics and Computer Science,
Wrocław University of Technology
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław, Poland

Abstract. By *k-disturbed sequence* we mean a sequence obtained from a sorted sequence by changing the values of at most k elements. We present an algorithm for single-hop radio networks that sorts a k-disturbed sequence of length n (where each station stores single key) in time $4n + k \cdot (\lceil \lg k \rceil^2 + \lceil \lg(n-k+1) \rceil + 6\lceil \lg k \rceil) - 2$ with energetic cost $3 \cdot \lceil \frac{(\lceil \lg k \rceil + 1)(\lceil \lg k \rceil + 2)}{2\lfloor n/k \rfloor} \rceil + 4 \cdot \lceil \frac{\lceil \lg k \rceil}{\lfloor n/k \rfloor} \rceil + 10$. If $\frac{(\lceil \lg k \rceil + 1)(\lceil \lg k \rceil + 2)}{2} + \lceil \lg k \rceil \leq \lfloor n/k \rfloor$ then the energetic cost is bounded by 14.

1 Introduction

By *k-disturbed sequence* we mean a sequence obtained from a sorted sequence by changing the values of at most k elements. We consider the problem of sorting k-disturbed sequence of length n, for a relatively small value of k. We call it *k-correction problem.* Our model of computation is a single hop radio network. Such network consists of n stations $s_0, \ldots, s_{n-1}$ communicating with each other by exchanging short radio messages. The stations are synchronized. Time is divided into *slots.* Within a single time slot a single message can be broadcast. During each time slot each station is either listening or sending or idle. If it is sending or listening then it dissipates a unit of energy. We assume that the stations are powered by batteries. Therefore we want to minimize *energetic cost* of the algorithm, i.e. the maximum over all stations of non-idle time slots. We consider *single hop* network: Each station is in the range of any other station. If two or more stations send messages simultaneously, then a *collision* occurs. Since our algorithm is avoiding collisions, we do not need to state precisely what happens during the collisions and whether they are detectable.

Algorithm for k-correction can be applied whenever we want to keep sorted a sequence of values that can change infrequently. As one example consider the following scenario: The stations are deployed in some area and equipped with sensors measuring some relatively stable value (e.g. temperature or air pressure). We want to keep the measured values sorted. Thus our algorithm can be invoked periodically, if we expect that the number of values changed within each period is substantially lower than n.

* This work has been supported by MNiSW grant N N206 1842 33

The naive solution of our problem is to sort the input sequence using one of the existing sorting algorithms (e.g. [9], [4]). However, the energetic cost of such algorithms is $\Omega(\log n)$ and the time of energetically efficient algorithms is $\Omega(n \cdot \log n)$. The direct simulation of comparator sorting networks (e.g. [1], [2]), where each comparator between two positions is simulated in two time slots by the corresponding two stations, leads to quite efficient sorting algorithms. The problem of k-correction has also been studied for the comparator networks (e.g [7], [10]). However, they are practically efficient for very small (sub-polynomial) and fixed values of k. On the other hand, the algorithms for radio networks can be adaptive, and we can use the fact that each station can notice the change of its own key. The value of k in our algorithm can be arbitrary (even as large as $n-1$, although the normal merge-sort (i.e. [4]) is more efficient for k close to n). The costs of the algorithm (time and energetic cost) are adapted to the actual value of k. As long as $\frac{(\lceil \lg k \rceil+1)(\lceil \lg k \rceil+2)}{2} + \lceil \lg k \rceil \leq \lfloor n/k \rfloor$, the energetic cost of our algorithm is bounded by 14 and for arbitrary k it is bounded by $3 \cdot \lceil \frac{(\lceil \lg k \rceil+1)(\lceil \lg k \rceil+2)}{2\lfloor n/k \rfloor} \rceil + 4 \cdot \lceil \frac{\lceil \lg k \rceil}{\lfloor n/k \rfloor} \rceil + 10$. Moreover, the algorithm uses only a constant number of variables within each station.

2 Preliminaries

Each station s_i initially stores a key in its local variable $oldKey[s_i]$. Variable $oldIdx[s_i]$ is the index of $oldKey[s_i]$ in the sorted sequence of keys. (The indexes are numbered from 0 to $n-1$.) The station s_i also stores a new value of its key in $newKey[s_i]$ which is either equal to or different from $oldKey[s_i]$. The task of each s_i is to compute $newIdx[s_i]$ which is the index of $newKey[s_i]$ in the sorted sequence of the new keys. (The keys remain in their originating stations. We just compute their positions in the sorted sequence.)

In this paper "lg" denotes "$\log_2$". Whenever we define a permutation π of $\{0,\ldots,n-1\}$, π^{-1} denotes the permutation reverse to π.

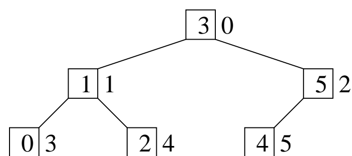

Fig. 1. The tree of T_6.

Let T_m be a balanced binary tree consisting of m nodes, such that the last level of T_m is filled from left to right (see Figure 1). We define two ways of indexing of the nodes of T_m:

y-indexing: The y-index of the root is zero and, for each node with y-index y, the indexes of its left and the right son are $l(y) = 2(y+1) - 1$ and

$r(y) = 2(y+1)$, respectively. For y-index y the level of y in the tree is $lev(y) = \lfloor \log_2(y+1) \rfloor$ and, if $y > 0$, then the y-index of its parent is $p(y) = \lfloor (y-1)/2 \rfloor$.

x-indexing: The x-indexing is obtained by numbering the nodes in the in-order with the numbers $0, \ldots, m-1$.

On Figure 1: The x-index is printed inside each node and the y-index – to the right of the node. *Binary search ordering* bso_m is a permutation of $\{0, \ldots, m-1\}$,

```
function bso_m(x)
if x ∉ {0,...,m−1} then return x; (* x outside domain *)
y ← 0; (* y-index of the current node. We start from the root. *)
x_1 ← ls(m); (* x-index of the root is the size of its left subtree *)
m_1 ← m; (* size of the subtree of current node *)
while x_1 ≠ x do
    if x < x_1 then
        y ← l(y); (* y-index of the left child *)
        m_1 ← ls(m_1); (* size of the left subtree *)
        x_1 ← x_1 − m_1 + ls(m_1); (* x-index of the left child *)
    else
        y ← r(y); (* y-index of the right child *)
        m_1 ← rs(m_1); (* size of the right subtree *)
        x_1 ← x_1 + ls(m_1) + 1; (* x-index of the right child *)
return y;
```

Algorithm 1: Permutation bso_m

such that $bso_m(x) = y$ if and only if the node with x-index x has the y-index equal to y. We define some functions and algorithm that can be used for efficient computation of *bso*. The height of T_m is $h(m) = \lceil \log_2(m+1) \rceil$. The size of full binary tree of height h is $fs(h) = 2^h - 1$. Thus the number of missing leaves on the last level of T_m is $ml(m) = fs(h(m)) - m$. Let $ls(m)$ be the size of the left subtree of the root of T_m. If $m \leq 1$ then $ls(m) = 0$ else $ls(m) = fs(h(m)-1) - \max\{0, ml(m) - 2^{h(m)-2}\}$. (The left subtree is full if there are at most $2^{h(m)-2}$ missing leaves in T_m.) Thus the size of the right subtree of the root of T_m is $rs(m) = m - 1 - ls(m)$. The position of y-index y within its level is $inlev(y) = y - fs(lev(y))$. Using these functions we can define efficient algorithm for computing *bso* (see Algorithm 1). Note that the **while** loop in Algorithm 1 iterates at most $h(m) \approx \lg m$ times and each iteration involves a constant number of computations of elementary functions. Thus the computation of *bso* is efficient.

In the remaining algorithms we also use functions $at(l, i) = fs(l) + i$ (the y-index of the ith node at level l) and $levsize_m(l)$ (the size of the lth level of T_m). Note that, for $0 \leq l < h(m) - 1$, $levsize_m(l) = 2^l$ and, for $l = h(m) - 1$, $levsize_m(l) = 2^l - ml(m)$.

3 Description of the algorithm

The network performs Algorithm 2. The general idea is to isolate the modified keys, sort them and merge them with the (sorted) sequence of the unmodified keys. We also try to balance the energetic costs among almost **all** the stations by forcing them to act as *virtual workers*.

```
begin
  split-and-count;
  (* all stations have learned k – the number of changed keys *)
  if k ≥ n/3 then
    apply sorting algorithm to compute new ranks of keys (e.g. the simple
    merge-sort from [4] without permuting the keys between the stations as
    is done in [6])
  else if k > 0 then
    assign-workers;
    sort;
    final-merge;
    Each s_i does: oldIdx[s_i] ← newIdx[s_i]; oldKey[s_i] ← newKey[s_i];
end
```

Algorithm 2: Correction algorithm.

Initially each station recognizes whether its key is changed. Then the stations perform the procedure split-and-count (Algorithm 3) that counts the number of changed keys and computes the initial position of each key either in the sequence of not changed keys (called *a-sequence*) or in the sequence of the changed keys (called *b-sequence*). Split-and-count is similar to the first phase of the counting sort algorithm ([3]). The ordering of the keys within a-sequence and within b-sequence is the same as in the original sorted sequence. Thus the a-sequence is already sorted.

Then the procedure assign-workers (Algorithm 4) assigns equal number of stations (*workers*) to each key (*b-key*) from the b-sequence. Each real station s simulates one *virtual station* vs that is used as a worker.

Next the b-sequence is sorted by a simple merge-sort algorithm ([4]), however the energy required by each b-key is balanced among all the workers (virtual stations) assigned to this key. This is done by the procedure sort (Algorithm 5). Sorting is done by merging neighboring blocks of sorted b-keys of length m into sorted blocks of length $2m$. We start with $m = 1$ (sorted singletons) and end-up with $m \geq k$. Procedure merge($[i_1, i_2]$, $[i_3, i_4]$) (Algorithm 6) merges the block of b-keys on positions $i_1,\ldots, i_2$ with the block of b-keys on positions $i_3,\ldots, i_4$. To merge the blocks each b-key from one block learns its rank in the other block and adds it to its index in the sorted sequence of the b-keys from its own block. In the procedure rank($[i_1, i_2]$,$\langle b_0, \ldots, b_{m-1}\rangle$,$d$) (Algorithm 8), each b-key from the block $[i_1, i_2]$ learns its rank in the block of keys stored in the stations

procedure split-and-count

Each s_i does (in parallel): **begin**

  $idx[s_i] \leftarrow oldIdx[s_i]$; $idx_a[s_i] \leftarrow NIL$; $idx_b[s_i] \leftarrow NIL$; $sum[s_i] \leftarrow 0$;

  $key[s_i] \leftarrow newKey[s_i]$;

  if $newKey[s_i] \neq oldKey[s_i]$ **then** $changed[s_i] \leftarrow 1$ **else** $changed[s_i] \leftarrow 0$;

end

for *time slot* $t \leftarrow 0$ **to** $n-2$ **do**

  Station s_i with $idx[s_i] = t$ does: **begin**

    if $changed[s_i] = 1$ **then** $idx_b[s_i] \leftarrow sum[s_i]$

    else $idx_a[s_i] \leftarrow idx[s_i] - sum[s_i]$;

    s_i sends msg, where $msg = sum[s_i] + changed[s_i]$;

  end

  Station s_j with $idx[s_j] = t+1$ receives msg and does: $sum[s_j] \leftarrow msg$;

in time slot $n-1$: **begin**

  Station s_i with $idx[s_i] = n-1$ does: **begin**

    if $changed[s_i] = 1$ **then** $idx_b[s_i] \leftarrow sum[s_i]$

    else $idx_a[s_i] \leftarrow idx[s_i] - sum[s_i]$;

    s_i sends msg, where $msg = sum[s_i] + changed[s_i]$;

  end

  Each station s_j receives msg and does $k[s_j] \leftarrow msg$; (* number of changed keys *)

end

Algorithm 3: Split and count.

procedure assign-workers

(* Each s_i has the same value $k[s_i]$ denoted by k. *)

(* The value $\lfloor n/k \rfloor$ is denoted by gs (*group size*) *)

for *time slot* $t \leftarrow 0$ **to** $k-1$ **do**

  The station s_i with $idx_b[s_i] = t$ sends msg, where $msg = newKey[s_i]$;

  Each s_j with $t \cdot gs \leq j < (t+1) \cdot gs$, does: **begin**

    s_j creates virtual station $vs_{t,t'}$, where $t' = j \bmod gs$;

    $vs_{t,t'}$ receives msg;

    $key[vs_{t,t'}] \leftarrow msg$; (* $vs_{t,t'}$ will be the t'th worker for tth changed key *)

    $rworker[vs_{t,t'}] \leftarrow 0$; (* initial index of current r-worker *)

    $iworker[vs_{t,t'}] \leftarrow gs - 1$; (* initial index of current i-worker *)

  end

Algorithm 4: Assigning workers.

procedure sort
Each virtual station does: $m \leftarrow 1$;
Each $vs_{i,j}$ with $j = iworker[vs_{i,j}]$ does: $idx[vs_{i,j}] \leftarrow 0$;
while $m < k$ **do**
  for $i \leftarrow 0$ **to** $\lfloor \frac{k}{2m} \rfloor - 1$ **do**
    merge($[2i \cdot m, (2i+1)m - 1]$, $[(2i+1)m, (2i+2)m - 1]$);
  if $k \bmod (2m) > m$ **then**
    merge($[i, i+m-1]$, $[i+m, k-1]$), where $i = \lfloor \frac{k}{2m} \rfloor \cdot 2m$;
  Each virtual station does: $m \leftarrow 2 \cdot m$;

Algorithm 5: Sorting.

procedure merge($[i_1, i_2]$, $[i_3, i_4]$)
begin
  (* Let b_i denote $vs_{i,j}$ with $iworker[vs_{i,j}] = j$. *)
  rank($[i_1, i_2]$, $\langle b_{i_3}, \ldots, b_{i_4} \rangle$, 0);
  rank($[i_3, i_4]$, $\langle b_{i_1}, \ldots, b_{i_2} \rangle$, 1);
  transfer-indexes($[i_1, i_2]$);
  transfer-indexes($[i_3, i_4]$);
end

Algorithm 6: Merging.

procedure transfer-indexes($[i_1, i_2]$)
(* For each $i \in [i_1, i_2]$, all $vs_{i,j}$ have the same value of $iworker[vs_{i,j}]$, denoted here by iw_i. Let $iw'_i = (iw_i - 1) \bmod gs$.*)
for *time slot* $t \leftarrow 0$ **to** $i_2 - i_1$ **do**
  Let $r = i_1 + t$; $vs_{r,iw}$ sends $msg = newIdx[vs_{r,iw}]$ to $vs_{r,iw'}$, and $vs_{r,iw'}$ does: $idx[vs_{r,iw'}] \leftarrow msg$.
Each $vs_{i,j}$, for $i_1 \leq i \leq i_2$, does: $iworker[vs_{i,j}] \leftarrow iw'$.

Algorithm 7: Transferring indexes.

$b_0, \ldots, b_{m_1}$. Each worker can act as an *i-worker* or as a *r-worker*. The task of an i-worker (respectively, r-worker) is to update the current index (respectively, rank) of its b-key. For *stability* of sorting and avoiding collisions of indexes, the parameter d (either zero or one) is used to decide whether the b-key from the block $[i_1, i_2]$ should be ranked before or after the equal b-keys from $b_0, \ldots, b_{m_1}$. Each b-key from $b_0, \ldots, b_{m_1}$ is broadcast only once (by its current i-worker).

```
procedure rank([i_1, i_2],⟨b_0, ..., b_{m-1}⟩,d)
For i_1 ≤ i ≤ i_2, each vs_{i,j} with j = rworker[vs_{i,j}] does: rank[vs_{i,j}] ← 0.
for l ← 0 to h(m) − 2 do
    (* Let a_i be the vs_{i,j} with rworker[vs_{i,j}] = j *)
    lrank(l, ⟨a_{i_1}, ..., a_{i_2}⟩,⟨b_0, ..., b_{m-1}⟩,d);
    transfer-ranks([i_1, i_2]);
(* Let a_i be the vs_{i,j} with rworker[vs_{i,j}] = j *)
lrank(h(m) − 1, ⟨a_{i_1}, ..., a_{i_2}⟩,⟨b_0, ..., b_{m-1}⟩,d); (* on the last level of T_m *)
send-ranks-to-indexes([i_1, i_2]);
```

Algorithm 8: Ranking.

Each i-worker knows the index *idx* of is b-key in the sorted sequence of its block. These indexes are permuted by the bso_m an the keys are transmitted according to this permutation. Thus, the sorted sequence is transmitted level by level of the T_m-tree and each b-key (that knows its rank in the previously transmitted levels) has to listen only once to determine its rank (the *newRank*) on the currently transmitted level. In the T_m-tree, each level is interleaved with the sorted sequence consisting of all the elements above this level. The last level of T_m may be non-full, therefore in the computation of *newRank* in the lrank for the last level we are considering this case by adding the size of the level to the ranks of the b-keys ranked after it.

```
procedure lrank(l,⟨a_0, ..., a_{l-1}⟩, ⟨b_0, ..., b_{m-1}⟩,d);
for time slot r ← 0 to levsize_m(l) − 1 do
    The unique b_j with bso_m(idx[b_j]) = at(l, r) sends msg = key[b_j];
    Each a_i with rank[a_i] = r listens and does: begin
        if (key[a_i] ≤ msg and d = 0) or (key[a_i] < msg and d = 1) then
            newRank[a_i] ← 2r;
        else
            newRank[a_i] ← 2r + 1;
    end
    (* The following may happen if l is the last level of the tree T_m of bso_m. *)
    Each a_i with rank[a_i] ≥ levsize_m(l) does:
    newRank[a_i] ← rank[a_i] + levsize_m(l);
```

Algorithm 9: Level ranking.

To avoid excessive energy loss in a single station, each level of the bso_m-tree is transmitted in a separate procedure lrank (Algorithm 9) and between the lranks the task of ranking on the next level is transferred from the current r-worker to the next r-worker by transfer-ranks (Algorithm 10). After the lrank for the last level, the current r-worker knows the rank of its b-key in the whole transmitted sequence and sends it to the corresponding i-worker (in the procedure send-ranks-to-indexes – Algorithm 11). The i-worker can now compute the index of the b-key in the merged sequence.

procedure transfer-ranks($[i_1, i_2]$)
(* For each $i \in [i_1, i_2]$, all $vs_{i,j}$ have the same value of $rworker[vs_{i,j}]$, denoted here by rw_i. Let $rw'_i = (rw_i + 1) \bmod gs$.*)
for *time slot* $t \leftarrow 0$ **to** $i_2 - i_1$ **do**
    Let $i = i_1 + t$; vs_{i,rw_i} sends $msg = newRank[vs_{i,rw_i}]$ to vs_{i,rw'_i}, and vs_{i,rw'_i} does: $rank[vs_{i,rw'_i}] \leftarrow msg$.
Each $vs_{i,j}$, for $i_1 \leq i \leq i_2$, does: $rworker[vs_{i,j}] \leftarrow rw'_i$.

Algorithm 10: Transferring ranks.

procedure send-ranks-to-indexes($[i_1, i_2]$)
(* For each $i \in [i_1, i_2]$, all $vs_{i,j}$ have the same value of $rworker[vs_{i,j}]$ and of $iworker[vs_{i,j}]$, denoted here by rw_i and iw_i respectively. Let $rw'_i = (rw_i + 1) \bmod gs$.*)
for *time slot* $t \leftarrow 0$ **to** $i_2 - i_1$ **do**
    Let $i = i_1 + t$;
    vs_{i,rw_i} sends $msg = newRank[vs_{i,rw_i}]$ to vs_{i,iw_i}, and vs_{i,iw_i} does: $newIdx[vs_{i,iw_i}] \leftarrow idx[vs_{i,iw_i}] + msg$.
Each $vs_{i,j}$, for $i_1 \leq i \leq i_2$, does: $rworker[vs_{i,j}] \leftarrow rw'_i$.

Algorithm 11: Sending and adding ranks to indexes.

As soon as each i-worker knows the index of its b-key in the merged sequence, these indexes are transferred to the next i-workers in the procedures transfer-indexes.

After the b-sequence is sorted, the procedure final-merge (Algorithm 12) merges the two sorted sequences: a-sequence with b-sequence. First we use the workers to perform ranking of each b-key in the a-sequence. Each station b_t learns the rank of $newKey[b_t]$ in the a-sequence by overhearing the t-th time slot of the procedure send-ranks-to-indexes concluding this ranking. Then (in the loop **A:**) each b-key learns its index in the sorted b-sequence (variable idx) and in the final output sequence (variable $newIdx$). In the loop **B:** each b-key learns the rank of its successor in the b-sequence. Thus each b-key knows whether it is the last b-key with the same rank. Then (in the loop **C:**) each a-key key_a is

informed by the last b-key key_b with the $rank$ equal to to the position of key_a in the a-sequence about its $rank$ in the b-sequence. Next (in the loop **D:**), each informed a-key informs its uninformed successors in the a-sequence about their rank in the b-sequence. Finally, each a-key computes its position in the sorted output sequence as the sum of its position in its own sequence and its rank in the other sequence. Here, each b-key is ranked before the equal or greater a-keys.

```
    procedure final-merge
    (* For 0 ≤ i < n − k, let a_i be the station s_j with idx_a[s_j] = i. For 0 ≤ i < k, let
    b_i be the s_j with idx_b[s_j] = i.*)
    Each a_i does: idx[a_i] ← idx_a[a_i];
    rank([0, k − 1],⟨a_0, . . . , a_{n−k−1}⟩,0);
    (* b_t listens to the message msg sent during the t-th time slot of
    send-ranks-to-indexes in the above rank procedure, and does: rank[b_t] ← msg *)
    (* Let v_i be the vs_{i,j} with iworker[vs_{i,j}] = j. *)
A:  for t ← 0 to k − 1 do
        v_t sends msg = newIdx[v_t] to b_t and b_t does:
        newIdx[b_t] ← msg; idx[b_t] ← msg − rank[b_t];
    The b_i with idx[b_i] = k − 1 does: last[b_i] ← TRUE
B:  for t ← k − 1 downto 1 do
        the b_i with idx[b_i] = t sends msg = rank[b_i];
        the b_j with idx[b_j] = t − 1 listens and does:
        if rank[b_j] ≠ msg then last[b_j] ← TRUE else last[b_i] ← FALSE;
    Each a_i does: if i = 0 then mov[a_i] ← 0 else mov[a_i] ← NIL;
C:  for t ← 0 to n − k − 1 do
        the b_i with rank[b_i] = t and last[b_i] = TRUE sends msg = idx[b_i];
        a_t listens and if it received msg then it does: mov[a_t] ← msg + 1;
D:  for t ← 0 to n − k − 2 do
        a_t sends msg = mov[a_t];
        If mov[a_{t+1}] = NIL then a_{t+1} listens and does: mov[a_{t+1}] ← msg;
    Each a_i does newIdx[a_i] ← idx_a[a_i] + mov[a_i];
```

Algorithm 12: Final merging of a-sequence with b-sequence.

4 Analysis of Complexity

Time: Time of split-and-count is $t_1 = n$. Time of assign-workers is $t_2 = k$. Let t_3 be the time of sort. Let $t_M(m_1, m_2)$ denote the time of merging two sequences with lengths m_1 and m_2. $t_M(m_1, m_2) = t_R(m_1, m_2) + t_R(m_2, m_1) + m_1 + m_2$, where $t_R(m', m'')$ is time of ranking m' elements in the sequence of length m'' and $m_1 + m_2$ is the time of transfer-indexes. Note that $t_R(m', m'') = m'' + h(m'') \cdot m'$, where m'' is the total time spent in all lranks and $h(m'') \cdot m'$ is the remaining time (spent in transfer-ranks and send-ranks-to-indexes). Thus $t_3 \leq \sum_{i=0}^{\lceil \lg k \rceil - 1} \lceil k/2^{i+1} \rceil \cdot$

$t_M(2^i, 2^i) \leq k(\lceil\lg k\rceil^2 + 6\lceil\lg k\rceil)$ (see Appendix A). Let t_4 be the time of final-merge. $t_4 = t_R(k, n-k) + k + (k-1) + (n-k) + (n-k-1) = (n-k) + k \cdot \lceil\lg(n-k+1)\rceil + k + (k-1) + (n-k) + (n-k-1) = k \cdot \lceil\lg(n-k+1)\rceil + 3n - k - 2$. Thus the total time is: $t_1 + t_2 + t_3 + t_4 \leq 4n + k \cdot (\lceil\lg k\rceil^2 + \lceil\lg(n-k+1)\rceil + 6\lceil\lg k\rceil) - 2$.

Energy: In the procedure split-and-count each station broadcasts once and listens at most twice. After that each station acts either as a-station or a b-station. In assign-workers each b-station broadcasts once and each station listens once. In the procedures sort and final-merge, the energy used for each b-key key_t with idx_b equal t is balanced among the $gs = \lfloor n/k \rfloor$ virtual stations $vs_{t,0}, \ldots, vs_{t,gs-1}$. By current r-worker (respectively, i-worker) we mean the virtual station $vs_{t,j}$ with $rworker[vs_{t,j}] = j$ (respectively, $iworker[vs_{t,j}] = j$). The tasks of r-worker and of i-worker are transferred in round-robin fashion in opposite directions. Initially, $vs_{t,0}$ becomes the first r-worker of its group and $v_{t,gs-1}$ becomes the first i-worker of its group. Each time $vs_{t,i}$ becomes r-worker it listens at most twice (once in transfer-ranks and once in lrank) and broadcasts once (either in the next transfer-ranks or in send-ranks-to-indexes). Then the $vs_{t,(i+1) \bmod gs}$ becomes the next r-worker. The key_t requires α r-workers, where $\alpha \leq \sum_{i=0}^{\lceil\lg k\rceil} h(2^i) = \frac{(\lceil\lg k\rceil+1)(\lceil\lg k\rceil+2)}{2}$. (The last component of the sum comes from the rank in final-merge.) Each time $v_{t,i}$ becomes i-worker it listens at most twice (once in transfer-indexes and once in send-ranks-to-indexes) and broadcasts twice (once in rank and once either in the next transfer-indexes or in the loop **A:** of final-merge). Then the $v_{t,(i-1) \bmod gs}$ becomes the next i-worker. The key key_t requires β i-workers, where $\beta \leq \lceil\lg k\rceil$. We split the energy used by the virtual station $vs_{t,i}$ into two components: the energy used as r-worker: e_r and the energy used as i-worker: e_i. Since $vs_{t,i}$ becomes r-worker (respectively, i-worker) once $\lfloor n/k \rfloor$ times, we have $e_r \leq 3 \cdot \lceil \frac{\alpha}{\lfloor n/k \rfloor} \rceil$ and $e_i \leq 4 \cdot \lceil \frac{\beta}{\lfloor n/k \rfloor} \rceil$. If $\beta + \alpha \leq \lfloor n/k \rfloor$ then the energy used by $vs_{t,i}$ is at most 4, otherwise it can be bounded by $e_r + e_i$. In the procedure final-merge, each b-station listens at most three times (once in rank, once in loop **A:**, and once in loop **B:**) and broadcasts at most twice (once in loop **B:** and once in loop **C:**). Each a-station listens at most twice (once in loop **C:** and once in loop **D:**) and broadcasts at most twice (once in rank and once in loop **D:**). Let e_a (respectively, e_b) denote the energy used by a-station (respectively, b-station). Thus, $e_a = 5$ and $e_b = 7$ and the total energy used by each station can be bound by: $3 + \max\{e_a, e_b\} + e_i + e_r \leq 10 + 3 \cdot \lceil \frac{(\lceil\lg k\rceil+1)(\lceil\lg k\rceil+2)}{2\lfloor n/k \rfloor} \rceil + 4 \cdot \lceil \frac{\lceil\lg k\rceil}{\lfloor n/k \rfloor} \rceil$. Additionally, if $\frac{(\lceil\lg k\rceil+1)(\lceil\lg k\rceil+2)}{2} + \lceil\lg k\rceil \leq \lfloor n/k \rfloor$ then the energetic cost is bounded by 14.

5 Conclusions and Final Remarks

The following theorem concludes the discussion of previous sections:

Theorem 1. *There exists an algorithm that sorts a k-disturbed sequence of length n distributed among n stations of single-hop radio network in time at most $4n + k \cdot (\lceil\lg k\rceil^2 + \lceil\lg(n-k+1)\rceil + 6\lceil\lg k\rceil) - 2$ with energetic cost at most*

$3 \cdot \lceil \frac{(\lceil \lg k \rceil + 1)(\lceil \lg k \rceil + 2)}{2 \lfloor n/k \rfloor} \rceil + 4 \cdot \lceil \frac{\lceil \lg k \rceil}{\lfloor n/k \rfloor} \rceil + 10$. *If* $\frac{(\lceil \lg k \rceil + 1)(\lceil \lg k \rceil + 2)}{2} + \lceil \lg k \rceil \leq \lfloor n/k \rfloor$ *then the energetic cost of the algorithm is bounded by* 14.

□

Consider a scenario in which the algorithm is invoked periodically. The stations with largest indexes have greater chance of not being used as virtual worker (if $n \bmod \lfloor n/k \rfloor \neq 0$). Since the the indexes of the stations are independent of the indexes of the keys in the sequence, we can try to balance this chance among all stations by re-indexing them between the periods (for example, by adding one modulo n to each index).

Our algorithm is designed for reliable network (i.e. each transmitted message is received by each listening station with probability one). It would be interesting to consider k-correction in unreliable networks. (Sorting algorithm for such networks has been proposed in [6]).

References

1. Ajtai, M., Komlós, J., Szemerédi, E.: Sorting in c log n parallel steps. Combinatorica 3, 1-19 (1983)
2. Batcher, K.E.: Sorting networks and their applications. Proceedings of 32nd AFIPS, pp. 307-314, 1968.
3. Gębala, M., Kik, M.: Counting-Sort and Routing in a Single Hop Radio Network. In: Kutyłowski, M., Cichoń, J., Kubiak, P. (eds.) ALGOSENSORS 2007. LNCS, vol. 4837, pp. 138-149. Springer, Heidelberg (2008)
4. Kik, M.: Merging and Merge-sort in a Single Hop Radio Network. In: Wiedermann, J., Tel, G., Pokorný, J., Bieliková, M., Stuller, J. (eds.) SOFSEM 2006. LNCS, vol. 3831, pp. 341-349. Springer, Heidelberg (2006)
5. Kik, M.: Sorting Long Sequences in a Single Hop Radio Network. In: Královic, R., Urzyczyn, P. (eds.) MFCS 2006. LNCS, vol. 4162, pp. 573-583. Springer, Heidelberg (2006)
6. Kik, M: Ranking and Sorting in Unreliable Single Hop Radio Network. In: D. Coudert et al. (Eds.): ADHOC-NOW 2008, LNCS 5198, pp. 333-344, 2008. Springer-Verlag Berlin Heidelberg (2008)
7. Kik, M., Kutyłowski, M., Piotrów, M.: Correction Networks, ICPP 1999, pp. 40-47,
8. Nakano, K.: An Optimal Randomized Ranking Algorithm on the k-channel Broadcast Communication Model. In: ICPP 2002, pp. 493-500 (2002)
9. Singh, M. and Prasanna, V.K.: Energy-Optimal and Energy-Balanced Sorting in a Single-Hop Sensor Network. PERCOM, March 2003.
10. Stachowiak, G.: Fibonacci Correction Networks. In: Magnús M. Halldórsson (eds.): SWAT 2000, LNCS 1851, pp. 535–548, 2000. Springer, Heidelberg 2000.

A Analysis of time complexity of the procedure sort

$$
\begin{aligned}
t_3 &\leq \sum_{i=0}^{\lceil \lg k \rceil - 1} \lceil k/2^{i+1} \rceil \cdot t_M(2^i, 2^i) = \sum_{i=0}^{\lceil \lg k \rceil - 1} \lceil k/2^{i+1} \rceil \cdot (2 \cdot t_R(2^i, 2^i) + 2 \cdot 2^i) \\
&= \sum_{i=0}^{\lceil \lg k \rceil - 1} \lceil k/2^{i+1} \rceil \cdot (2 \cdot (2^i + h(2^i) \cdot 2^i) + 2 \cdot 2^i) = \sum_{i=0}^{\lceil \lg k \rceil - 1} \lceil k/2^{i+1} \rceil \cdot 2^{i+1}(2 + h(2^i))
\end{aligned}
$$

Since $h(2^i) = i + 1$, we have:

$$
t_3 = \sum_{i=0}^{\lceil \lg k \rceil - 1} \lceil k/2^{i+1} \rceil \cdot 2^{i+1}(3 + i) \leq \sum_{i=0}^{\lceil \lg k \rceil - 1} (k/2^{i+1} + 1) \cdot 2^{i+1}(3 + i) = \sum_{i=0}^{\lceil \lg k \rceil - 1} (k + 2^{i+1})(3 + i).
$$

For $0 \leq i \leq \lceil \lg k \rceil - 1$, we have $2^{i+1} < k$. Thus

$$
\begin{aligned}
t_3 &\leq \sum_{i=0}^{\lceil \lg k \rceil - 1} 2 \cdot k \cdot (3 + i) = 6k\lceil \lg k \rceil + 2k \cdot \sum_{i=0}^{\lceil \lg k \rceil - 1} i = 6k\lceil \lg k \rceil + 2k \cdot \frac{(\lceil \lg k \rceil - 1)\lceil \lg k \rceil}{2} \\
&\leq k(\lceil \lg k \rceil^2 + 6\lceil \lg k \rceil).
\end{aligned}
$$