Abstract. We propose an algorithm for merging two sorted sequences of length \( k \cdot m \) stored in two sequences of \( m \) stations of single-hop single-channel radio network, where each station stores a block of \( k \) consecutive elements. The time and energetic cost of this algorithm are \( 6m \cdot k + 8m - 4 \) and \( 8k + 4\log_2(m + 1) + 6 \), respectively. This algorithm can be applied for sorting a sequence of length \( N = k \cdot n \) in a network consisting of \( n \) stations with memory limited by \( \Theta(k) \) words. For \( k = \Omega(\log n) \), the energetic cost of such sorting is \( O(k \cdot \log n) \) and the time is \( O(N \log n) \). Moreover, the constants hidden by the big “Oh” are reasonably small, to make the algorithm attractive for practical applications.

1 Introduction

We consider the following problem: A sequence of length \( N = n \cdot k \) is distributed among \( n \) stations of a single-hop radio network. Each station stores \( k \) elements of the sequence. The length of the sequence significantly exceeds the number of stations (i.e. \( k = \Omega(\log n) \)). We want to sort this sequence. The stations are synchronized. Time is divided into slots. Within a single time slot a single message can be broadcast. We consider single-hop network: Message broadcast by any station can be received by any other station. A single message contains \( O(\max\{B, \log N\}) \) bits, where \( B \) is the number of bits of a single key (i.e. element of the input sequence). (Typically \( B = \Theta(\log(N)) \).)

For any station, sending or listening in a single time slot requires a unit of energy. The main goal is to minimize energetic cost of the algorithm, i.e. the maximal energy dissipated by any single station. This prolongs the lifetime of the battery powered stations of the network. We also assume that each station can store \( \Theta(k) \) words of \( \max\{B, \log N\} \) bits each.

The sorting algorithms proposed in [9] and [4] assume that each station stores only one key and it is not evident how to adopt them to the case when each station stores \( k \) keys. The sorting algorithm in [9] is based on energetically balanced selection [8] and obtains energetic cost \( O(\log n) \). On the other hand, [4] contains description of simple merge-sort with energetic cost \( O(\log^2 n) \) that due to its low constants and simplicity is attractive for practical applications.

There exists an algorithm [6] that sorts \( n \) elements in time \( O(n) \) with energetic cost of broadcasting \( O(1) \): Each station \( s_i \) listens the keys broadcast by remaining stations

* Supported by the European Union within the 6th Framework Programme under contract 001907 (DELIS).
and computes the rank $r$ of its own key. In the $r$th slot of the next stage $s_i$, broadcasts its key to $s_r$. This algorithm can be immediately adopted for sorting $k \cdot n$ keys in time $O(kn)$ with energetic cost of broadcasting $O(k)$. However the energetic cost of listening in this algorithm would be $\Theta(k \cdot n)$.

A comparator network sorting sequences of length $n$ can be directly simulated on a single-hop network of $n$ stations: each comparator is simulated in two consecutive time slots, when two endpoints of the comparator exchange their values. The time of such an algorithm (in single channel network) is two times the number of comparators, and the energetic cost is not greater than two times the depth of the network. (Actually, it is twice the maximal number of comparisons performed by a single station.)

We can transform such algorithm into an algorithm for sorting sequences of size $n \cdot k$ using the following standard method for comparator networks (see [5], chapter 5.3.4, exercise 38): Each of the $n$ elements of the sequence is replaced by a sorted block of $k$ elements and each of the comparisons of two elements is replaced by sorting (actually merging) of the corresponding two groups. The energetic cost and time of such operation for each involved station is $2k$: It has to broadcast all its keys and receive all keys of the other station.

For example, the AKS sorting network [1] can be transformed into (impractical) algorithm sorting sequences of length $k \cdot n$ in time $O(k \cdot n \log n)$ and with energetic cost $O(k \cdot n \log n)$ and the Batcher networks [2] can be transformed into algorithms with time $O(k \cdot n \log^2 n)$ and energetic cost $O(k \cdot \log^2 n)$.

In this paper we present a practical algorithm (based on the simple merging algorithm from [4]) that merges two sequences of length $k \cdot n$ stored in two sequences of $n$ stations in time $O(k \cdot n)$ and with energetic cost $O(n \log n)$. For the case $k > \log n$, the energetic cost is $O(k)$. This algorithm can be used for merge-sorting in time $O(k \cdot n \log n)$ and energetic cost $O(k \log n)$. This is asymptotically equivalent to the algorithm obtained from AKS network, but the constants involved are much lower.

## 2 Preliminaries

The network consists of $n$ stations. Initially we have a sequence of $n \cdot k$ keys evenly distributed among the stations: Each station $s_i$ stores a (sorted) sequence of $k$ keys in the table $key[s_i][1 \ldots k]$. (This table is extended in both directions by $key[s_i][0]$ and $key[s_i][k+1]$.) By interval of $s_i$ we mean the interval $[key[s_i][1], key[s_i][k]]$. After sorting, all the elements in $key[s_i][1 \ldots k]$ are less than all the elements in $key[s_{i+1}][1 \ldots k]$, for $1 \leq i < n$.

For simplicity of description we assume that all the keys are pairwise distinct. We ignore the cost of internal operations inside single stations and for the sake of readability we do not optimize them.

## 3 Merging

We start with an algorithm for merging two sorted sequences of length $k \cdot m$ (called $a$-sequence and $b$-sequence) stored in two sequences of stations $(a_1, \ldots, a_m)$ (i.e. $a$-
stations) and \( \langle b_1, \ldots, b_m \rangle \) (i.e. \( b \)-stations), respectively. The result will be a sorted sequence of length \( 2k \cdot m \) stored in the sequence of stations \( \langle a_1, \ldots, a_m, b_1, \ldots, b_m \rangle \).

### 3.1 An Overview of the Algorithm

In the procedure \( \text{Try-Ranking}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle) \) (defined in the following section) each station \( a_i \) computes the ranks of all its keys in the \( b \)-sequence unless its interval is split by the interval of some \( b_j \), and each station \( b_j \) computes the ranks of all its keys in the \( a \)-sequence if its interval contains at least one endpoint of the interval of some \( a_j \). (I.e. Its interval neither splits the interval of any \( a_j \) nor is disjoint with all intervals \( a_j \).) All the uncomputed ranks are computed by the symmetrical procedure \( \text{Try-Ranking}(\langle b_1, \ldots, b_m \rangle, \langle a_1, \ldots, a_m \rangle) \). As soon as the ranks are known, the final position of each key in the merged sequence is also known. We finish by performing simple permutation routing, where the destination of each key is its final position in the merged sequence.

### 3.2 Technical Details

We repeat some technical definitions from [4]. Let \( T_m \) denote a balanced binary tree consisting of the nodes \( 1, \ldots, m \): If \( m = 2^k - 1 \), for some integer \( k > 0 \), then \( T_m \) is a complete binary tree. If \( m = 2^k - 1 - l \), for some positive integer \( l < 2^k - 1 \), then the \( l \) rightmost nodes on the last level are missing. Thus the shape of \( T_m \) is identical to the shape of binary heap containing \( m \) elements (see [3]). However, the nodes are placed in \( T_m \) in the in-order order (i.e. for each node \( x \) the nodes in its left subtree are less than \( x \) and the nodes in its right subtree are greater than \( x \)). By \( l(m, x) \) (respectively \( r(m, x) \)), for \( 1 \leq x \leq m \), we denote the left (respectively right) child of node \( x \) in \( T_m \). (A non-existing child is represented by \( \text{NIL} \).) By \( p(m, x) \) we denote the index of node \( x \) in \( T_m \) in heap-order ordering, which corresponds to the index of \( x \) in an array representation of \( T_m \) treated as a heap. (I.e. the heap-order index of the root is 1, then the nodes on the second level are indexed from left to right, then on the third level, and so on.) We also assume that \( p(m, \text{NIL}) = \text{NIL} \). An example of \( T_m \) for \( m = 6 \) is given in Figure 1.

Note that the height (number of levels) of \( T_m \) is \( \min \{ k : 2^k - 1 \geq m \} = \lfloor \log_2(m + 1) \rfloor \) (where “\( \log \)” denotes “\( \log_2 \)”). The algorithm uses several procedures defined below.

![Fig. 1. Tree \( T_6 \). Right to the nodes are their heap-order indexes.](image)
procedure Init\((a_1, \ldots, a_m)\)
begin
  \(a_1\) does: \(\text{key}[a_1][0] \leftarrow -\infty\);
  \(a_m\) does: \(\text{key}[a_m][k + 1] \leftarrow +\infty\);
  for time slot \(i \leftarrow 1\) to \(m - 1\) do
    station \(a_i\) broadcasts \(x\), where \(x = \text{key}[a_i][k]\);
    station \(a_{i+1}\) listens and does: \(\text{key}[a_{i+1}][0] \leftarrow x\);
  end
end

Algorithm 1: Procedure Init.

In procedure Init (Algorithm 1) each station is informed about the maximal key stored by its predecessor and the minimal key stored by its successor.

Each station \(a_i\) contains additional variables \(l\text{Partner}[a_i]\), \(r\text{Partner}[a_i]\), \(l\text{Rank}[a_i]\), and \(r\text{Rank}[a_i]\). \(l\text{Partner}[a_i]\) and \(r\text{Partner}[a_i]\) can store either \(\text{NIL}\) or a triple \((x, f, l)\), where \(x\) is an index of some \(b\)-station \(b_x\) and \(f, l\) are the endpoints of the interval of \(b_x\). For each \(a_i\), in procedure Find-Partners (Algorithm 2), for \(k_1 = \text{key}[a_i][1]\), we compute in \(l\text{Partner}[a_i]\) the index and the endpoints of the station \(b_x\) such that \(k_1\) is in the interval of \(b_x\). If \(k_1\) is ranked between any two intervals of \(b_x\) and \(b_{x+1}\), then \(l\text{Partner}[a_i] = \text{NIL}\) and its final rank in the other sequence (equal \(k \cdot x\)) is computed in \(l\text{Rank}[a_i]\). (Note that if \(x = 0\) then \(b_x\) does not exist, and if \(x = m\) then \(b_{x+1}\) does not exist.) We make analogous computations for \(\text{key}[a_i][k]\) and variables \(r\text{Partner}[a_i]\) and \(r\text{Rank}[a_i]\). The computations for each endpoint of \(a_i\) are independent. \(a_i\) receives only those messages that can influence one of its endpoints. Each time the endpoints of the interval of some \(b\)-station are received, the local procedure Update (Algorithm 3) is invoked for the endpoint of \(a_i\) that can be influenced. Update uses its two initial parameters to update the variables referenced by the remaining parameters. Finally, each station \(a_i\) uses the values \(l\text{Partner}[a_i]\), \(r\text{Partner}[a_i]\), \(l\text{Rank}[a_i]\) and \(r\text{Rank}[a_i]\), to compute \(\text{split}[a_i]\). The variable \(\text{split}[a_i]\) becomes true if and only if the interval of some \(b_j\) is properly contained in the interval of \(a_i\).

Each station \(s\) contains a table \(\text{rank}[s][1 \ldots k]\). Initially, \(\text{rank}[a_i][r] = \text{rank}[b_i][r] = \text{NIL}\), for all \(1 \leq i \leq m\) and \(1 \leq r \leq k\). We want to compute in each \(\text{rank}[a_i][r]\) the rank of \text{key}[a_i][r] in \(b\)-sequence, and vice versa (i.e. in each \(\text{rank}[b_i][r]\) the rank of \text{key}[b_i][r] in \(a\)-sequence). By the rank of \(x\) we mean the number of keys less than \(x\) in the other sequence. Procedure Try-Ranking (Algorithm 4) computes the ranks in the \(b\)-stations that have partners among the \(a\)-stations and in each \(a_j\) that has \(\text{split}[a_j] = \text{false}\). Procedure Rank (Algorithm 6) computes ranks in all the stations, and Merge (Algorithm 7) uses the ranks for computing final positions of the keys in the sorted sequence and routes them to their destinations.
procedure Find-Partners((a₁, ..., aₘ), (b₁, ..., bₘ))
begin
    Each station aᵢ does: begin
        lTimer[aᵢ] ← rTimer[aᵢ] ← 1;
        lRank[aᵢ] ← rRank[aᵢ] ← 0;
        lPartner[aᵢ] ← rPartner[aᵢ] ← NIL;
        split[aᵢ] ← false;
    end
end

for time slot d ← 1 to m do
    let x be such that p(m, x) = d; (* d is heap-order index of x *)
    station bₓ broadcasts (f, l), where f = key[bₓ][1] and l = key[bₓ][k];
    each station aᵢ with d = lTimer[aᵢ] or d = rTimer[aᵢ] listens and does: begin
        if d = lTimer[aᵢ] then
            Update(hₓ, f, l, key[aᵢ][0], lTimer[aᵢ], lRank[aᵢ], lPartner[aᵢ]);
        if d = rTimer[aᵢ] then
            Update(hₓ, f, l, key[aᵢ][k], rTimer[aᵢ], rRank[aᵢ], rPartner[aᵢ]);
    end
end

Each station aᵢ does: begin
    let lP = lPartner[aᵢ], rP = rPartner[aᵢ], lR = lRank[aᵢ], and rR = rRank[aᵢ].
    if (lP = NIL ∧ lR < rR) or
    (lP = (x, ...) ∧ rP = (x', ...) ∧ x + 1 < x') or
    (lP = (x, ...) ∧ rP = NIL ∧ x · k < rR) or
    (lP = NIL ∧ rP = (x', ...) ∧ lR < (x' - 1) · k) then
        (* aᵢ is split by some bⱼ *)
        split[aᵢ] ← true
    end
end

Algorithm 2: Procedure Find-Partners.

procedure Update((x, f, l), key, Timer, Rank, Partner)
(* Timer, Rank, and Partner are references to variables *)
begin
    if f < key < l then
        Partner ← (x, f, l);
        Timer ← NIL;
    else if key < f then
        Timer ← p(m, l(m, x)); (* heap-order index of left child of x *)
    else if l < key then
        Timer ← p(m, r(m, x)); (* heap-order index of right child of x *)
        Rank ← x · k; (* key is preceded by at least x · k keys in the other sequence *)
end

Algorithm 3: Procedure Update.
procedure Try-Ranking((a_1, \ldots, a_m), (b_1, \ldots, b_m))
begin
  Init((a_1, \ldots, a_m));
  Find-Partners((a_1, \ldots, a_m), (b_1, \ldots, b_m));
  for i ← 1 to m do
    for r ← 1 to k do
      In time slot 2((i - 1) \cdot k + r) - 1:
        b_i broadcasts \(v = key[b_i][r]\);
        each \(a_j\) with \(lPartner[a_j] = (i, f, l)\) or \(rPartner[a_j] = (i, f, l)\) listens
        and does:
        if \(split[a_j] = false\) then
          forall 1 ≤ s ≤ k do
            if v < key[a_j][s] then
              rank[a_j][s] ← (i - 1) \cdot k + r; (* index of v in the b-sequence *)
        end
      end
    end
  for r ← 1 to k do
    In time slot 2((i - 1) \cdot k + r):
    Let \((j, s)\) be the (at most one) pair, such that \(lPartner[a_j] = (i, f, l)\) or
    \(rPartner[a_j] = (i, f, l)\) and:
    - \((1 \le s \le k \land key[a_j][s - 1] < v < key[a_j][s])\), or
    - \((s = k + 1 \land key[a_j][s - 1] < v \le l < key[a_j][s])\).
    (* i.e. \(key[a_j][s]\) is the successor of v in the a-sequence and, for \(s = k + 1\),
    \(b_i\) is not a partner of \(a_{j+1}\). *)
    If such \((j, s)\) exists, then \(a_j\) broadcasts \(y\), where \(y = (j - 1) \cdot k + s - 1\).
    \(b_i\) listens and does:
      if there was a message \(y\) then
        rank[b_i][r] ← y; (* index of key[a_j][s - 1] in the a-sequence *)
      end
  end
Each \(a_i\) does internally: Rank-Unsplit(a_i);
end

Algorithm 4: Procedure Try-Ranking.

procedure Rank-Unsplit(a_j)
begin
  if \(split[a_j] = false\) then
    if \(lPartner[a_j] = rPartner[a_j] = NIL\) then
      for r ← 1 to k do
        rank[a_j][r] ← lRank[a_j];
    else if \(lPartner[a_j] = NIL\) then
      last ← max\{i \mid key[a_j][i] < f\}, where \(rPartner[a_j] = (x, f, l)\);
      for r ← 1 to last do
        rank[a_j][r] ← lRank[a_j];
    end
  end
end

Algorithm 5: Procedure Rank-Unsplit.
procedure Rank$(a_1, \ldots, a_m, b_1, \ldots, b_m)$
begin
  Each $s \in \{a_1, \ldots, a_m, b_1, \ldots, b_m\}$ does internally: begin
    for $r \leftarrow 1$ to $k$ do $rank[s][r] \leftarrow NIL$
  end
  Try-Ranking$((a_1, \ldots, a_m), (b_1, \ldots, b_m))$;
  Try-Ranking$((b_1, \ldots, b_m), (a_1, \ldots, a_m))$;
end

Algorithm 6: Procedure Rank.

procedure Merge$(a_1, \ldots, a_m, b_1, \ldots, b_m)$
begin
  Rank$(a_1, \ldots, a_m, b_1, \ldots, b_m)$;
  Each station $a_i$ does internally:
    for $r \leftarrow 1$ to $k$ do $idx[a_i][r] \leftarrow (i-1) \cdot k + r + rank[a_i][r]$;
  Each station $b_i$ does internally:
    for $r \leftarrow 1$ to $k$ do $idx[b_i][r] \leftarrow (i-1) \cdot k + r + rank[b_i][r]$;
  (* for $1 \leq i \leq m$ let $c_i = a_i$ and $c_{m+i} = b_i$ *)
  for time slot $t \leftarrow 1$ to $2m \cdot k$ do
    station $c_t$ with $idx[c_t][r] = t$ broadcasts $(k)$, where $k = key[c_t][r]$;
    (* Let $t' = \lceil (t-1)/k \rceil + 1$ and $r = t - (t' - 1) \cdot k$ *)
    station $c_{t'}$ listens and does: $new[c_{t'}][r] \leftarrow k$;
  Each station $c_i$, for $1 \leq r \leq k$: $key[c_i][r] \leftarrow new[c_i][r]$;
end

Algorithm 7: Procedure Merge.
3.3 Correctness of Merge

Lemma 1. For each endpoint $e$ of the interval of each $a_i$, $\text{Find-Partners}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle)$ either computes its partner $\langle x, f, l \rangle$ such that $f = \text{key}[b_x][0] < e < \text{key}[b_x][k] = l$ or (if such $b_x$ does not exist) its rank in $b$-sequence. Variable $\text{split}[a_i]$ becomes true if and only if the interval of some $b$-station is inside the interval of $a_i$.

Consider arbitrary $a_i$. Let $e = \text{key}[a_i][1]$ (i.e. the left endpoint of $a_i$). Let $t_1, \ldots, t_r$ be the initial consecutive values (different form NIL) of $l\text{Timer}[a_i]$ during the computation. Let $x_1, \ldots, x_r$, be such that $t_j = p(m, x_j)$. Let $\langle f_j, t_j \rangle = \langle \text{key}[b_x][1], \text{key}[b_x][k] \rangle$. Note that, for each $j < r$, either $e < f_j$ or $l_j < e$ and $x_j+1$ is the child of $x_j$ in $T_m$ on the same side that $e$ is to the interval of $b_x$. The ordering of intervals of $b$-stations is the same as the ordering of their indexes. Hence $x_1, \ldots, x_r$ is the path in $T_m$ that should be followed by the ordinary bisection algorithm searching for position of $e$ among the intervals of $b$-stations. Thus, every time $l\text{Rank}[a_i]$ and $l\text{Timer}[a_i]$ are properly updated by $\text{Update}$. If (after $\text{Find-Partners}$) $l\text{Partner}[a_i] \neq \text{NIL}$ then its correctness follows from the code of $\text{Update}$. The reasoning for the right endpoint of $a_i$ is the same.

After the variables $l\text{Partner}, l\text{Rank}, r\text{Partner}$ and $r\text{Rank}$ have been correctly computed in $a_i$, the correctness of the computation of $\text{split}[a_i]$ follows from the observation that some $b$-station is inside the interval of $a_i$ if and only if $a_i$ has no partners and the ranks of its endpoints are different, or $a_i$ has two partners with indexes more distant than one, or $a_i$ has one partner and the other endpoint of the interval of $a_i$ is not ranked immediately before or immediately after the keys of this partner.

We have to show that all values in all tables rank are properly computed by $\text{Rank}$. We show that each of these tables is computed by at least one of the two procedures $\text{Try-Ranking}$ in $\text{Rank}$. We say that station $a_i$ is split in $\text{Try-Ranking}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle)$ if it ends up with $\text{split}[a_i] = \text{true}$. We say that station $b_i$ is splitting in $\text{Try-Ranking}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle)$ if exists $a_j$ such that $\text{key}[a_j][1] < \text{key}[b_i][1]$ and $\text{key}[b_i][k] < \text{key}[a_j][k]$ (i.e. $b_i$ splits $a_j$). We say that $b_i$ is avoided in $\text{Try-Ranking}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle)$ if the interval of $b_i$ does not intersect interval of any $a_j$.

Lemma 2. Procedure $\text{Try-Ranking}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle)$ correctly computes all ranks in unsplit $a$-stations and in unsplitting $b$-stations that are avoided. In the remaining stations the tables rank are not modified.

Let $a_j$ be unsplit. After $\text{Find-Partners}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle)$ there are four possible cases:

Case 1: $l\text{Partner}[a_j] = r\text{Partner}[a_j] = \text{NIL}$. In this case interval of $a_j$ is disjoint with all intervals of $b$-stations and the value $r = l\text{Rank}[a_j] = r\text{Rank}[a_j]$ is the rank of each key of $a_j$ in the $b$-sequence. Procedure $\text{Rank-Unsplit}(a_j)$ fills $\text{rank}[a_j][1 \ldots k] with r.

Case 2: $l\text{Partner}[a_j] = (x, f, l)$, for some $x$ and $f < l$, and $r\text{Partner}[a_j] = \text{NIL}$. In the fragment following Find-Partners in $\text{Try-Ranking}$, $a_j$ listens to all keys broadcast in increasing order by $b_x$ during the $x$-th iteration of the external “for” loop and adjusts all the ranks of its own greater keys. Thus, after the last key of $b_x$ is broadcast, all the ranks in $a_j$ are correct.
Lemma 6. If \( \text{lrPartner}[a_j] = \text{NIL} \) and \( r\text{Partner}[a_j] = \langle x, f, l \rangle \), for some \( x \) and \( f < l \).

In this case \( \text{lrRank}[a_j] \) is the proper rank of all keys of \( a_j \) less than \( f \). This part of \( \text{rank}[a_j][1 \ldots k] \) is adjusted in \( \text{Rank-Unsplit} \). The remaining ranks are adjusted during the \( x \)-th iteration of the external “for” loop after \( \text{Find-Partners} \).

Case 4: \( \text{lrPartner}[a_j] = \langle x, f, l \rangle \) and \( r\text{Partner}[a_j] = \langle x + 1, f', l' \rangle \), for some \( x \) and \( f < l < f' < l' \). The ranks of the keys of \( a_j \) that are less than \( f' \) are computed during the \( x \)-th iteration of the external “for” loop after \( \text{Find-Partners} \). Remaining ranks in \( a_j \) are computed during the \((x + 1)\)-st iteration of this loop.

Let \( b_i \) be unsplitting and not avoided. Then the the interval of \( b_i \) contains the endpoints of all the intervals of \( a \)-stations that intersect the interval of \( b_i \). Hence, after \( \text{Find-Partners} \), \( b_i \) is a partner of all those \( a \)-stations and, for each key \( v \) of \( b_i \) there exists some (unique) \( a_j \) with \( \text{lrPartner}[a_j] = \langle i, \ldots \rangle \) or \( r\text{Partner}[a_j] = \langle i, \ldots \rangle \) that contains successor of \( v \) or the last element of \( a \)-sequence in the interval of \( b_i \). This station is responsible for answering to the message \( \langle v \rangle \) broadcast by \( b_i \). (Note that \( a_j \) may be split.)

Let \( a_i \) be split. The instructions “if \( \text{split}[a_j] = \text{false} \)” in \( \text{Try-Ranking} \) and \( \text{Rank-Unsplit} \) prevent modifications of \( \text{rank}[a_j][1 \ldots k] \).

Let \( b_i \) be splitting or avoided. Then \( b_i \) is partner of any \( a_j \) and no one answers to its messages broadcast in the \( i \)-th iteration of the external “for” loop after \( \text{Find-Partners} \). Only those answers could have caused modifications of \( \text{rank}[b_i] \). □

Lemma 3. If \( a_i \) is split in \( \text{Try-Ranking}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle) \), then it is unsplitting and not avoided in \( \text{Try-Ranking}(\langle b_1, \ldots, b_m \rangle, \langle a_1, \ldots, a_m \rangle) \).

There is some \( b_j \) with its interval properly contained in the interval of \( a_i \). The intervals of \( b_1, \ldots, b_m \) are disjoint. Thus the interval of \( a_i \) can not be contained in any one of them (i.e. \( a_i \) is unsplitting) and intersects the interval of \( b_j \) (i.e. \( a_i \) is not avoided). □

Lemma 4. If \( b_i \) is splitting in \( \text{Try-Ranking}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle) \), then it is unsplit in \( \text{Try-Ranking}(\langle b_1, \ldots, b_m \rangle, \langle a_1, \ldots, a_m \rangle) \).

The interval of \( b_i \) is properly contained in the interval of some \( a_j \). Thus the procedure \( \text{Find-Partners}(\langle b_1, \ldots, b_m \rangle, \langle a_1, \ldots, a_m \rangle) \) ends up with \( \text{lrPartner}[b_i] = r\text{Partner}[b_i] = \langle j, \text{key}[a_j][1], \text{key}[a_j][k] \rangle \) and with \( \text{split} = \text{false} \). □

Lemma 5. If \( b_i \) is avoided in \( \text{Try-Ranking}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle) \), then it is unsplit in \( \text{Try-Ranking}(\langle b_1, \ldots, b_m \rangle, \langle a_1, \ldots, a_m \rangle) \).

The interval of \( b_i \) is disjoint with each \( a_j \). Thus none interval of \( a_j \) can be properly contained in the interval of \( b_i \). □

Lemma 6. Procedure \( \text{Rank} \) correctly computes all ranks in all stations.

By Lemmas 2, 3, 4, and 5 all the ranks in all the stations are correctly computed in at least one of \( \text{Try-Ranking}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle) \) or \( \text{Try-Ranking}(\langle b_1, \ldots, b_m \rangle, \langle a_1, \ldots, a_m \rangle) \). (The second \( \text{Try-Ranking} \) does compute all the ranks missing after the first \( \text{Try-Ranking} \) and doesn’t overwrite any computed ranks with wrong values.) □

Lemma 7. \( \text{Merge} \) correctly merges \( a \)-sequence with \( b \)-sequence.
Rank computes the rank of each key in the other sequence. Thus the final position of each key in the merged sequence is the sum of its position in its own sequence and its rank in the other sequence. These are exactly the values computed in tables idx. Since all the keys are pairwise distinct, there is exactly one value $t$ in all tables idx, for each $1 \leq t \leq 2k \cdot m$, and in each iteration of the "for" loop exactly one message is broadcast. (We do not need the reservation phase mentioned in simple routing protocol in [7], since the destinations are positions in the sequence – not indexes of the stations.)

3.4 Estimations of Time and Energetic Costs.

To make the comparison with other algorithms more fair, we assume that a single message may contain either a single key or a single index of $\lceil \lg(N) \rceil$ bits, where $N$ is total number of keys. Therefore we replace each message $(f, l)$ broadcast in Find-Partners by two messages $(f)$ and $(l)$ broadcast in two consecutive time slots. Let $T_M$ denote the time of Merge. $T_M = T_R + 2m \cdot k$, where $T_R$ is the time of Rank. $T_R = 2T_TR$, where $T_TR$ is the time of Try-Ranking. $T_TR = T_I + T_FP + 2m \cdot k$, where $T_I$ is time of Init and $T_FP$ is time of Find-Partners. $T_I = 2m - 2$. In Find-Partners, each $b_i$ broadcasts once both endpoints of its interval. Hence, $T_FP = 2m$.

Thus $T_TR = (2m - 2) + (2m) + 2m \cdot k = 2m \cdot k + 4m - 2$, and $T_R = 4m \cdot k + 8m - 4$, and $T_M = 6m \cdot k + 8m - 4$.

We estimate separately the energetic cost of listening $L_M$ and of sending $S_M$ of Merge. This is more informative in the case when sending requires more energy than listening. However, we assume that the total energetic cost of Merge is $E_M = S_M + L_M$. Thus $S_M = S_R + k$ and $L_M = L_R + k$ (where $S_R$ and $L_R$ are the respective costs of Rank), since in the "for" loop each $c_i$ listens $k$ times and broadcasts each of its keys exactly once. $S_R = S_{TR,a} + S_{TR,b}$ and $L_R = L_{TR,a} + L_{TR,b}$, where $S_{TR,a}$ and $L_{TR,a}$ (respectively, $S_{TR,b}$ and $L_{TR,b}$) are the costs for $a$-stations (respectively, $b$-stations) in Try-Ranking. $S_{TR,a} = S_I + 2k$ (where $S_I$ is cost of sending in Init), since some $a_i$ may be obliged to respond to all keys $(v)$ broadcast by its both partners. $L_{TR,a} = L_I + L_{FP,a} + 2k$ (where $L_I$ and $L_{FP,a}$ are the listening costs for $a$-stations in Init and Find-Partners), since each $a_i$ has to listen to all keys $(v)$ broadcast by its at most two partners. $S_{TR,b} = S_{FP,b} + k$ (where $S_{FP,b}$ is cost of sending in Find-Partners), since each $b_i$ broadcasts each its key as $(v)$. $L_{TR,b} = k + L_{FP,b}$, since each $b_i$ listens to each its message $(v)$. $S_{FP,b} = 2$, since each $b_i$ broadcasts its $(f, l)$ only once. $L_{FP,a} = 4\lceil \lg(m + 1) \rceil$, since each timer, after each updating, becomes the heap-order index of a node on the next level of $T_m$ or NIL. Hence each $a_i$ listens to $(f, l)$ at most twice on each level of $T_m$. $S_{FP,a} = 0$ and $L_{FP,b} = 0$, since $a$-stations do not broadcast and $b$-stations do not listen in Find-Partners. It is obvious that $S_I = L_I = 2$.

Thus $S_{TR,a} = 2k + 2$, $L_{TR,a} = 2k + 4\lceil \lg(m + 1) \rceil + 2$, $S_{TR,b} = k + 2$, $L_{TR,b} = k$, $S_R = 3k + 4$, $L_R = 3k + 4\lceil \lg(m + 1) \rceil + 2$, $S_M = 4k + 4$, and $L_M = 4k + 4\lceil \lg(m + 1) \rceil + 2$.

Thus the total energy of Merge is $E_M = 8k + 4\lceil \lg(m + 1) \rceil + 6$.

Further Improvements. Since, each message $(f, l)$ broadcast in Find-Partners contains the keys that are memorized by all interested $a$-stations, $b$-stations do not need repeat
sending them in the following “for” loops in Try-Ranking. This reduces the time of Try-Ranking by $2m$ and the sending energy of each $b$-station and listening energy of each $a$-station by 2. Thus the energetic cost of Merge is reduced by 4 and its time is reduced by $4m$. We have proven the following theorem:

**Theorem 1.** There exists algorithm merging two sorted sequences of length $k \cdot m$, divided into consecutive blocks of size $k$ stored in two sequences of $m$ stations, in time $6m \cdot k + 4m - 4$ with energetic cost $8k + 4[\lg(m + 1)] + 2$.

For comparison consider the Batcher comparator network for merging two sequences of length $m$ (either bitonic or odd-even merge [2]). It contains $\frac{m \lg m}{2}$ comparators. Thus the time of merging $a$-sequence with $b$-sequence with the adaptation of this network, as described in Section 1, requires $\approx m \lg m \cdot k$ time slots.

The energetic cost of the algorithm obtained from the Batcher network is $\approx 2k \lg m$, since the depth of the Batcher merging network is $\approx \lg m$.

For example, for $m = 2^{10} = 1024$, the time and energetic cost of our algorithm are $6144 \cdot k + 4092$ and $8k + 46$, respectively. For the adaptation of Batcher networks the time and energetic cost are $\approx 10240 \cdot k$ and $\approx 20 \cdot k$.

4 Sorting.

For simplicity, let $n$ be power of two. We can treat any sequence of length $N = n \cdot k$ as $n$ sorted sequences of length $k$. We merge each pair of consecutive sorted sequences into single sorted sequence. We repeat this operation $\lg n$ times to obtain a single sorted sequence of length $n$. Let $T_M(m)$ and $E_M(m)$ be the time and energetic cost of merging two sequences of length $m \cdot k$, placed in $m$ stations each. Then the time and energetic cost of our sorting procedure are $T_S(n, k) = \sum_{i=0}^{\lfloor \lg n \rfloor - 1} n \cdot T_M(2^i)/2^i$ and $E_S(n, k) = \sum_{i=0}^{\lfloor \lg n \rfloor - 1} E_M(2^i)$. If we apply our merging algorithm, then $T_S(n, k) \leq (3k + 2)n \lg n$ and $E_S(n, k) \leq (8k + 2)\lg n + \sum_{i=0}^{\lfloor \lg n \rfloor - 1} 4i = (8k + 2)\lg n + 2(\lg n + 1)(\lg n)$.

On the other hand the energetic cost and time for adaptations of Batcher algorithms are $\approx k \lg^2 n$ and $\approx \frac{1}{2}kn \lg^2 n$.

In merge-sort we can mix our merging algorithm with other merging or sorting algorithms (such as Batcher algorithms) that are more efficient for shorter subsequences. The proper choice depends on both parameters $n$ and $k$.

**Remark.** A simulation implemented in Java of the merging procedure described in this paper can be found at [10].

**References**