Sorting Long Sequence in a Single Hop Radio Network

Marcin Kik

2005
Abstract. In this paper we propose algorithm for merging two sorted sequences of length \( k \cdot m \) stored in blocks of size \( k \) in \( m \) stations of single-hop single-channel radio network. The time and energetic cost of this algorithm are \( 6m \cdot k + 8m - 4 \) and \( 8k + 4\lfloor \lg(m + 1) \rfloor + 6 \), respectively. This algorithm can be applied for sorting a sequence of length \( k \cdot n \) in a network consisting of \( n \) stations. For \( k = \Omega(\lg n) \), the energetic cost of such sorting is \( O(k \cdot \lg n) \) and the time is \( O(k \cdot n) \). 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 \( n \cdot k \) is evenly distributed among the \( n \) stations of a single-hop radio network, where the the length of the sequence significantly exceeds the number of stations (i.e. \( k = \Omega(\lg n) \)). We want to sort this sequence. The stations within a network 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 \( O(\max\{B, \lg(n \cdot k)\}) \) bits, where \( B \) is the number of bits of a single key. (Typically \( B = \Theta(\lg(n \cdot k)) \).) Sending or listening in a single time slot requires unit of energetic cost. The main goal is to minimize energetic cost of the algorithm, i.e. the maximal energy dissipated by any single station. This prolongs lifetime of the stations powered by batteries. We also assume that each station can store \( \Theta(k) \) words of \( \max\{B, \lfloor \lg(n \cdot k) \rfloor \} \) bits each.

The sorting algorithms proposed in [7] and [3] assume that each station stores only a single key and it is not not evident how to adopt them to the case when each station stores \( k \) keys. The sorting algorithm in [7] is based on energetically balanced selection [6] and obtains energetic cost \( O(\lg n) \). On the other hand, [3] contains description of simple merge-sort with energetic cost \( O(\lg^2 n) \) that due to its low constants and simplicity is attractive for practical applications. The result of this paper is based on this algorithm.

There exists an algorithm [5] that sorts \( n \) elements in time \( O(n) \) with energetic cost of broadcasting \( O(1) \) and can be immediately adopted for sorting \( k \cdot n \) keys in

---

* Partially supported by the European Union within the 6th Framework Programme under contract 001907 (DELIS).
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 networks 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 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 number of comparators connected to a single horizontal wire of the network.)

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 [4], chapter 5.3.4, exercise 38): Each of the $n$ elements of the sequence is replaced by an (internally 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 keys of its own block and receive all keys of the other block.

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

In this paper we present a practical algorithm (based on the simple merging algorithm from [3]) 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 $O(\max\{k, \lg n\})$. For the case $k > \lg n$, the energetic cost is $O(k)$. This algorithm is then used for merge-sorting in time $O(k \cdot n \lg n)$ and energetic cost $O(k \cdot \lg n)$. This is asymptotically equivalent to the algorithm obtained from AKS network, but the constants involved are much lower.

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 range 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.

## 2 Merging

We repeat some technical definitions from [3]. 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 leaves are missing. The nodes are placed in $T_m$ in the inorder 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 NIL.) By $p(m, x)$ we denote the index of node $x$ in $T_m$ in preorder ordering. (I.e. the
preorder 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 \( \min\{k : 2^k - 1 \geq m\} = \lceil \log_2(m + 1) \rceil \) (where “\( \log \)” denotes “\( \log_2 \)”).

![Tree T6](image)

**Fig. 1.** Tree \( T_6 \). Right to the nodes are their preorder indexes.

We start with algorithm for merging two sorted sequences of length \( k \cdot m \) (called a-sequence and b-sequence) stored in two sequences of stations \( \langle a_1, \ldots, a_m \rangle \) (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 \( \langle a_1, \ldots, a_m, b_1, \ldots, b_m \rangle \).

Procedure \textit{Init} informs each station about the greatest key stored by its predecessor and about the smallest key stored by its successor.

**Algorithm 1:** Procedure \textit{Init}.

```
procedure Init((a_1, \ldots, a_m))
begin
    a_1 does: key[a_1][0] \gets -\infty;
    a_m does: key[a_m][k + 1] \gets +\infty;
    for time slot i \from 1 to m - 1 do
        station a_i broadcasts (x), where x = key[a_i][k];
        station a_{i+1} listens and does: key[a_{i+1}][0] \leftarrow x;
    for time slot i \from 1 to m - 1 do
        station a_{i+1} broadcasts (x), where x = key[a_{i+1}][1];
        station a_i listens and does: key[a_i][k + 1] \leftarrow x;
end
```

Each station \( a_i \) contains additional variables \( lPartner[a_i], rPartner[a_i], lRank[a_i], \) and \( rRank[a_i] \). In Procedure \textit{FindPartners} (Algorithm 2), for each key \( x = key[a_i][1] \), we compute in \( lPartner[a_i] \) the index of the station \( b_j \) such that \( x \) is in the range of \( b_j \) (together with the endpoints of the range of \( b_j \)). If \( x \) is ranked between blocks \( b_j \) and \( b_{j+1} \), then its final rank in the other sequence is \( lRank[a_i] = k \cdot j \). (Note that if \( j = 0 \) then \( b_j \) does not exist, and if \( j = m \) then \( b_{j+1} \) does not exist.) We make analogous computations for each \( key[a_i][k] \) and variables \( rPartner[a_i] \) and \( rRank[a_i] \). In local procedure \textit{Update} (Algorithm 3) station \( a_i \) uses its two initial parameters to update the variables referenced by the remaining parameters. Finally, each station \( a_i \) uses the values \( lPartner[a_i], rPartner[a_i], lRank[a_i] \) and \( rRank[a_i] \), to compute \( \text{split}[a_i] \).
The variable $\text{split}[a_j]$ becomes true if and only if the range of some $b_j$ is properly contained in the range of $a_j$. (i.e. $a_j$ has no partners and the ranks of its range’s endpoints are different, or $a_j$ has both partners with indexes more distant than one, or $a_j$ has one partner and the other endpoint of the range of $a_j$ is not ranked immediately before or immediately after the keys of this partner.)

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 lesser than $x$ in the other sequence. Procedure $\text{TryRanking}$ (Algorithm 4) computes the

```
procedure FindPartners((a_1, \ldots, a_m), (b_1, \ldots, b_m))
begin
  Each station $a_i$ does: begin
    lTimer$[a_i] \leftarrow rTimer[a_i] \leftarrow 1$;
    lRank$[a_i] \leftarrow rRank[a_i] \leftarrow 0$;
    lPartner$[a_i] \leftarrow rPartner[a_i] \leftarrow \text{NIL}$;
    split$[a_i] \leftarrow \text{false}$;
  end
  for time slot $d \leftarrow 1 \text{ to } m$ do
    let $x$ be such that $p(m, x) = d$; (* $d$ is preorder index of $x$ *)
    station $b_x$ broadcasts $(f, l)$, where $f = \text{key}[b_x][1]$ and $l = \text{key}[b_x][k]$;
    each station $a_i$ with $d = l\text{Timer}[a_i]$ or $d = r\text{Timer}[a_i]$ listens and does: begin
      if $d = l\text{Timer}[a_i]$ then
        _ Update$(x, f, l), \text{key}[a_i][0], l\text{Timer}[a_i], l\text{Rank}[a_i], l\text{Partner}[a_i]$;
      if $d = r\text{Timer}[a_i]$ then
        _ Update$(x, f, l), \text{key}[a_i][k], r\text{Timer}[a_i], r\text{Rank}[a_i], r\text{Partner}[a_i]$;
    end
  end
  Each station $a_i$ does: begin
    Let lP = lPartner$[a_i], rP = rPartner[a_i], lR = lRank[a_i], and
    rR = rRank[a_i].
    if (lP = rP = \text{NIL} \land lR < rR) or
      (lP = (x, \ldots) \land rP = (x', \ldots) \land x + 1 < x') or
      (lP = (x, \ldots) \land rP = \text{NIL} \land x \cdot k < rR) or
      (lP = \text{NIL} \land rP = (x', \ldots) \land lR < (x' - 1) \cdot k) then
      _ split$[a_i] \leftarrow \text{true}$
    end
end
```

Algorithm 2: Procedure $\text{FindPartners}$. 

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 $\text{Rank}$ (Algorithm 6) computes ranks in all the stations, and $\text{Merge}$ (Algorithm 7) uses the ranks for computing final positions of the keys in the sorted sequence and routes them to their destinations.
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)); (* preorder index of left child of x *)
  else if l < key then
    Timer ← p(m, r(m, x)); (* preorder 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 TryRanking((a₁, ..., aₘ), (b₁, ..., bₘ))
begin
  Init((a₁, ..., aₘ));
  FindPartners((a₁, ..., aₘ), (b₁, ..., bₘ));
  for i ← 1 to m do
    for r ← 1 to k do
      In time slot 2((i − 1) · k + r) − 1:
        bᵢ broadcasts ⟨v⟩, where v = key[bᵢ][r];
        each aᵢ with lPartner[aᵢ] = ⟨i, f, l⟩ or rPartner[aᵢ] = ⟨i, f, l⟩ listens
        and does:
        if split[aᵢ] = false then
          forall 1 ≤ s ≤ k do
            if v < key[aᵢ][s] then
              rank[aᵢ][s] ← (i − 1) · k + r; (* index of v in the b-sequence *)
          In time slot 2((i − 1) · k + r):
            Let ⟨j, s⟩ be the (at most one) pair, such that lPartner[aⱼ] = ⟨i, f, l⟩ or
            rPartner[aⱼ] = ⟨i, f, l⟩ and (1 ≤ s ≤ k ∧ key[aⱼ][s − 1] < v <
            key[aⱼ][s]) ∨ (s = k + 1 ∧ key[aⱼ][s] < v ≤ l < key[aⱼ][s]).
            (* key[aⱼ][s] is the successor of v in a-sequence and, for s = k + 1, bᵢ is
            not a partner of aⱼ+1 *)
            If such (j, s) exists, then aⱼ broadcasts ⟨y⟩, where y = (j − 1) · k + s − 1.
            bᵢ listens and does: begin
              if there was a message ⟨y⟩ then
                rank[bᵢ][r] ← y; (* index of key[aⱼ][s − 1] in the a-sequence *)
            end
        Each aᵢ does internally: RankUnsplit(aᵢ);
    end
  end
end

Algorithm 4: Procedure TryRanking.
procedure RankUnsplit(a)
begin
if split[a] = false then
  if lPartner[a] = rPartner[a] = NIL then
    for r ← 1 to k do
      rank[a][r] ← lRank[a];
  else if lPartner[a] = NIL then
    last ← max{key[a][i] | i < f}, where rPartner[a] = h x; f; l i;
    for r ← 1 to last do
      rank[a][r] ← lRank[a];
  end
end

Algorithm 5: Procedure RankUnsplit.

procedure Rank((a_1, \ldots, a_m), (b_1, \ldots, b_m))
begin
  Each s ∈ {a_1, \ldots, a_m, b_1, \ldots, b_m} does internally: begin
    for r ← 1 to k do rank[s][r] ← NIL;
  end
  TryRanking((a_1, \ldots, a_m), (b_1, \ldots, b_m));
  TryRanking((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 ← 1 to k do idx[a_i][r] ← (i - 1) \cdot k + r + rank[a_i][r];
  Each station b_i does internally:
  for r ← 1 to k do idx[b_i][r] ← (i - 1) \cdot k + r + rank[b_i][r];
  (* for 1 ≤ i ≤ m let c_i = a_i and c_{n+i} = b_i *)
  for time slot t ← 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' = [(t - 1)/k] + 1 and r = t - (t' - 1) \cdot k *)
    station c_t listens and does: new[c_t][r] ← k;
    Each station c_i does, for 1 ≤ r ≤ k: key[c_i][r] ← new[c_i][r];
end

Algorithm 7: Procedure Merge.
We show that each of these tables is computed by at least one of the two procedures
\( f < l < f \) and \( x > l \) is the path in Lemma 2.

\[ \text{Lemma 1. } \text{FindPartners}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle), \text{for each endpoint } e \text{ of the range of each } a_i, \text{ either computes its rank in } b\text{-sequence or its partner } \langle x, f, l \rangle \text{ such that } f = \text{key}[b_2][0] < e < \text{key}[b_2][k] = l. \]

Consider arbitrary \( a_i \). The computations for its left endpoint and right endpoint are
independent. Consider one of these endpoints \( e \). If \( e \) ends up with partner different from
\( \text{NIL} \) then its correctness follows from the code of Update. Otherwise, let \( t_1, \ldots, t_r \) be
the values of its timer during the computation (different form \( \text{NIL} \)). Let \( x_1, \ldots, x_r \) be
such that \( t_i = p(m, x_i) \). Let \( \langle f_i, l_i \rangle = (\text{key}[b_x][0], \text{key}[b_x][1]) \). Note that, for each \( i \),
either \( e < f_i \) or \( e > l_i \) and each next \( x_i \) is on the lower level of \( T_m \). Hence \( x_1, \ldots, x_r \)
is the path in \( T_m \) that should be followed by the ordinary bisection algorithm, and each
time \( e > l_i \) the rank of \( e \) is properly updated. \( \square \)

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
TryRanking in \( \text{Rank} \). We say that station \( a_i \) is split in \( \text{TryRanking}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle) \) if it ends up with split[\( a_i \)] = true. We say that station \( b_i \) is splitting
in \( \text{TryRanking}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle) \) if exists \( a_j \) such that \text{key}[a_j][1] < key[b_i][1] and key[b_i][k] < key[a_i][k] (i.e. \( b_i \) splits \( a_j \)). We say that \( b_i \) is avoided
in \( \text{TryRanking}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle) \) if the range of \( b_i \) does not intersect range of
any \( a_j \).

\[ \text{Lemma 2. } \text{Procedure } \text{TryRanking}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle) \text{ correctly computes all ranks in unsplit stations } a_i \text{ and in unsplitting stations } b_i \text{ that are not avoided. In the remaining stations the tables rank are not modified.} \]

Let \( a_i \) be unsplit. Then, after \text{FindFriends}(\langle a_1, \ldots, a_m \rangle, \langle b_1, \ldots, b_m \rangle) \) there are
four possible cases:

\[ \text{Case 1: } \text{IPartner}[a_i] = rPartner[a_i] = \text{NIL}. \text{ In this case range of } a_i \text{ is disjoint}
\text{with all ranges of } b_j \text{ and the value } r = lRank[a_j] = rRank[a_j] \text{ is the rank of each key of } a_i \text{ in the } b\text{-sequence. Procedure RankUnsplit}(a_i) \text{ fills rank}[a_j][1 \ldots k] \text{ with } r. \]

\[ \text{Case 2: } \text{IPartner}[a_i] = \langle x, f, l \rangle, \text{ for some } x \text{ and } f < l, \text{ and } rPartner[a_i] = \text{NIL}. \text{ In the fragment following FindPartner in TryRanking, } a_i \text{ listens to all keys broadcast in increasing order by } b_x \text{ during the } x\text{-th iteration of the external "for" loop and adjusts all the ranks of its own greater keys. Thus, after the last key of } b_x \text{ is broadcast, all the ranks in } a_j \text{ are correct.} \]

\[ \text{Case 3: } \text{IPartner}[a_i] = \text{NIL and } rPartner[a_i] = \langle x, f, l \rangle, \text{ for some } x \text{ and } f < l. \text{ In this case } lRank[a_i] \text{ is the proper rank of all keys of } a_i \text{ less than } f. \text{ This part of rank}[a_i][1 \ldots k] \text{ is adjusted in RankUnsplit. The remaining ranks are adjusted during the } x\text{-th iteration of the external "for" loop after FindPartner.} \]

\[ \text{Case 4: } \text{IPartner}[a_i] = \langle x, f, l \rangle \text{ and } rPartner[a_i] = (x + 1, f', l'), \text{ for some } x \text{ and } f < l < f' < l'. \text{ The ranks corresponding to the keys of } a_i \text{ less than } f' \text{ are computed during the } x\text{-th iteration of the external for loop after FindPartner. Remaining ranks in } a_j \text{ are computed during the } (x + 1)\text{-st iteration of the external for loop after FindPartner.} \]
Let $b_i$ be unsplitting and not avoided. Then the the range of $b_i$ intersects some ranges of $a_j$. Hence, after $\text{FindFriends}((a_1, \ldots, a_m), (b_1, \ldots, b_m))$, for each key $v$ of $b_i$ there exists some (unique) $a_j$ with $l\text{Partner}[a_j] = \langle i, \ldots \rangle$ or $r\text{Partner}[a_j] = \langle i, \ldots \rangle$ that contains successor of $v$ or contains the last element in the range of $b_j$. This station is responsible for answering to the message $\langle v \rangle$ broadcast in $\text{TryRanking}((a_1, \ldots, a_m), (b_1, \ldots, b_m))$ by $b_j$. (Note that $a_j$ may be split.)

Let $a_i$ be split. The instructions "if split[$a_j$] = false" in $\text{TryRanking}$ and $\text{RankUnsplit}$ prevent modifications of $\text{rank}[a_i][1 \ldots k]$.

Let $b_i$ be splitting or avoided. Then $b_i$ is not partner of any $a_j$ and no one answers to its messages broadcast in the rth iteration of the external "for" loop after $\text{FindPartners}$. Only those answers could have caused modifications of $\text{rank}[b_i]$. □

**Lemma 3.** If $a_i$ is split in $\text{TryRanking}((a_1, \ldots, a_m), (b_1, \ldots, b_m))$, then it is unsplitting in $\text{TryRanking}((b_1, \ldots, b_m), (a_1, \ldots, a_m))$.

There is some $b_j$ with its range properly contained in the range of $a_i$. The ranges of $b_1, \ldots, b_m$ are disjoint. Thus the range of $a_i$ can not be contained in any one of them. □

**Lemma 4.** If $b_i$ is splitting in $\text{TryRanking}((a_1, \ldots, a_m), (b_1, \ldots, b_m))$, then it is unsplit in $\text{TryRanking}((b_1, \ldots, b_m), (a_1, \ldots, a_m))$.

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

**Lemma 5.** If $b_i$ is avoided in $\text{TryRanking}((a_1, \ldots, a_m), (b_1, \ldots, b_m))$, then it is unsplit in $\text{TryRanking}((b_1, \ldots, b_m), (a_1, \ldots, a_m))$.

The range of $b_i$ is disjoint with each $a_j$. Thus none range of $a_j$ can be properly contained in the range 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 $\text{TryRanking}((a_1, \ldots, a_m), (b_1, \ldots, b_m))$ or $\text{TryRanking}((b_1, \ldots, b_m), (a_1, \ldots, a_m))$. (The second $\text{TryRanking}$ does compute all the ranks missing after the first $\text{TryRanking}$ and doesn’t overwrite any computed ranks with wrong values.) □

**Lemma 7.** $\text{Merge}$ correctly merges a-sequence with b-sequence.

$\text{Rank}$ 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. □
2.2 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 \([\lfloor \log(k \cdot n) \rfloor]\) bits. Therefore we replace each message \((f, l)\) broadcast in \textbf{FindPartners} by two messages \((f)\) and \((l)\) broadcast in two consecutive time slots. Let \(T_M\) denote the time of \textbf{Merge}, \(T_M = T_R + 2m \cdot k\), where \(T_R\) is the time of \textbf{Rank}. \(T_R = 2T_{TR}\), where \(T_{TR}\) is the time of \textbf{TryRanking}. \(T_{TR} = T_l + T_{FP} + 2m \cdot k\), where \(T_l\) is time of Init and \(T_{FP}\) is time of \textbf{FindPartners}. \(T_l = 2m - 2\). In \textbf{FindPartners}, each \(b_i\) broadcasts once its \((f, l)\). By the fairness assumptions, \(T_{FP} = 2m\). Thus \(T_{TR} = (2m - 2) + (2m) + 2m \cdot k = 2m \cdot k + 4m - 2\), and \(T_M = 6m \cdot k + 8m - 4\), and \(T_M = 6m \cdot k + 8m - 4\).

We estimate separately the energetic cost of listening \(L_M\) and of sending \(S_M\) of \textbf{Merge}. This is more informative in the case when sending requires more energy than listening. However, we assume that the total energetic cost of \textbf{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 \textbf{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, \(S_{TR,b}\), \(L_{TR,b}\)) are the costs for \(a\)-stations (respectively, \(b\)-stations) in \textbf{TryRank}. \(S_{TR,a} = S_I + 2k\), where \(S_I\) is cost of sending in \textbf{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 \textbf{Init} and \textbf{FindPartners}, 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 \textbf{FindPartners}, 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. \(L_{FP,a} = 4 \lfloor \log(m + 1) \rfloor\), since each timer, after each updating, becomes the preorder 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 \textbf{FindPartners}. It is obvious that \(S_I = L_I = 2\). Thus \(S_{TR,a} = 2k + 2, L_{TR,a} = 2k + 4 \lfloor \log(m + 1) \rfloor + 2, S_{TR,b} = k + 2, L_{TR,b} = k, S_R = 3k + 4, L_R = 3k + 4 \lfloor \log(m + 1) \rfloor + 2, S_M = 4k + 4, and L_M = 4k + 4 \lfloor \log(m + 1) \rfloor + 2\). Thus the total energy of \textbf{Merge} is \(E_M = 8k + 4 \lfloor \log(m + 1) \rfloor + 6\).

Further Improvements. Since, each message \((f, l)\) broadcast in \textbf{FindPartners} 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 \textbf{TryRanking}. This reduces the time of \textbf{TryRanking} by \(2m\) and the sending energy of each \(b\)-station and listening energy of each \(a\)-station by \(2\). Thus the energetic cost of \textbf{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 \(8k + 4 \lfloor \log(m + 1) \rfloor + 2\).

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

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

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 $\approx 20 \cdot k$.

3 Sorting.

For simplicity, let $n$ be power of two. We can treat any sequence of length $n$ as $n$ sorted sequences of length one. We merge each pair of consecutive sorted sequences into single sorted sequence. We repeat this operation $\log n$ 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 size $m$. Then the time and energetic cost of our sorting procedure are

$$T_S(n) = \sum_{i=0}^{\log n - 1} n \cdot T_M(2^i)/2$$

and

$$E_S(n) = \sum_{i=0}^{\log n - 1} E_M(2^i).$$

If we apply our merging algorithm, then $T_S(n) \leq (3k + 2)n \log n$ and $E_S(n) \leq (8k + 2) \log n + \sum_{i=1}^{\log n} 4i = (8k + 2) \log n + 2(\log n + 1)(\log n)$.

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

Since our algorithm is based on merging, we can mix it with other merging or sorting algorithms (e.g. Batcher algorithms) that are more efficient for shorter subsequences.

Remark. A simulation in Java language of the merging procedure described in this paper can be found at [8].

References

8. Compendium of Large-Scale Optimization Problems. (DELIS, Subproject 3).
http://ru1.cti.gr/delis-sp3/