Succinct and explicit circuits for sorting and connectivity

Hamid Jahanjou Eric Miles Emanuele Viola

March 21, 2014

Abstract

We study which functions can be computed by efficient circuits whose gate connections are very easy to compute. We give quasilinear-size circuits for sorting whose connections can be computed by decision trees with depth logarithmic in the length of the gate description. We also show that NL has NC^2 circuits whose connections can be computed with constant locality.

*Supported by NSF grants CCF-0845003, CCF-1319206. Email: {hamid,enmiles,viola}@ccs.neu.edu

1 Introduction and our results

There are two prominent ways in which circuits for computing a given function $f : \{0,1\}^* \to \{0,1\}^*$ may be optimized. The first is to optimize parameters such as size and depth, a goal which permeates complexity theory and (up to the translation between running time and circuit size) algorithm research. The second is to make the circuits more uniform. That is, we seek simpler and simpler algorithms that on input 1^n produce the n-th circuit, which computes the restriction f_n of f to inputs of n bits. Popular uniformity conditions include P-uniformity, L-uniformity, and Logtime-uniformity. Restricting our attention to circuits of polynomial size, in the first case, we can output the n-th circuit in time $\text{poly}(n)$. In the second, we can output it in space $O(\log(n))$. Equivalently, given the $O(\log n)$-bit description of a gate in the circuit we can compute its type and neighbors in *linear* space $O(\log n)$. In the third, we can decide the type of a gate and whether a gate is input to another in linear time in their $O(\log n)$-bit description, on a Turing machine. There are several other uniformity conditions. For these, and for background on uniformity we refer the reader e.g. to the papers [Ruz81, All89, BIS90, BKLM12] and Vollmer's book [Vol99].

Typically, uniformity notions ask for a single algorithm that outputs the n-th circuit in the family. In this work we take a slightly different approach which is arguably also natural. We ask that for every n the n-th circuit has a "simple" description, but we are not as concerned with how this description changes from one n to another.

Definition 1. For a class D of functions (e.g., D = decision trees of depth $O(\log n)$), we call a family of circuits *D-explicit* if there is a function in D that given an index to a gate g in the n-th circuit outputs the type of g and the indices of the children of g, for every n.

In all of our results, a description of the function in D for the n-th circuit is computable in polynomial time from 1^n.

The authors recently show in [JMV13] that any L-uniform circuit family has an equivalent $\mathbf{NC}^0$-explicit circuit family. That is, given an index to a gate we can compute its type and the indices of its children by a function with constant locality: each output bit depends on $O(1)$ input bits only.

However, making circuits uniform or explicit is typically obtained at the expense of size or depth. For example, non-uniform or P-uniform circuits for approximate majority have depth 3 [Ajt83, Vio09], whereas for Logtime-uniformity the depth is larger [Ajt93]. Also, the above result in [JMV13] produces circuits of size at least 2^s and depth at least t when the machine describing the original circuit family uses space s and time t. In particular, the result only guarantees polynomial size and depth, no matter the size and depth of the original circuit.

In this work we suggest the study of *simultaneously* optimizing explicitness and efficiency. That is we wish to make the circuits as explicit as possible, ideally $\mathbf{NC}^0$-explicit, while keeping the size and/or the depth close to what is possible without uniformity restrictions. While we do not know of a general result, we show next how to come close to achieving this goal for two well-studied problems: sorting and directed connectivity. The notation $\widetilde{O}$ hides polylogarithmic factors.

Theorem 2 (Succinct, explicit sorting circuits)**.** *For all $m, n \in \mathbb{N}$, there is a D-explicit sorting circuit $C : (\{0,1\}^m)^n \to (\{0,1\}^m)^n$ where*

1. $|C| = \widetilde{O}(mn)$ *and* $D =$ *decision trees of depth* $O(\log\log(mn))$*, or*
2. $|C| = \widetilde{O}(mn)$ *and* $D =$ *circuits of size* $O(\log(mn))$ *and depth* $O(\log\log(mn))$*, or*
3. *for any* $b \leq \log(mn)$ *we can have* $|C| = \widetilde{O}(mn) \cdot (\log(mn)/b)^{O(b)}$ *and* $D =$ *decision trees of depth* $O(\log(\log(mn)/b))$*.*

Setting $b = \log(mn)/t$ in Item 3 we obtain decision trees of depth $O(\log t)$ and circuits of size $(mn)^{\log t/t}$. In particular, for any $\epsilon > 0$ we obtain $\mathbf{NC}^0$-explicit sorting circuits of size $(mn)^{1+\epsilon}$.

Independently of our work, Kowalski and Van Melkebeek proved a result similar to Item 1 in Theorem 2 with $\mathbf{AC}^0$ circuits of size $\mathrm{poly}(\log(mn))$ instead of decision trees of depth $O(\log\log(mn))$ (personal communication).

Theorem 3 (Succinct, explicit STCONN circuits)**.** *For all* $n \in \mathbb{N}$*, there is an* $\mathbf{NC}^0$*-explicit* $\mathbf{NC}^2$ *circuit* $C : \{0,1\}^{n^2} \to \{0,1\}$ *deciding directed connectivity in* n*-vertex graphs.*

The sorting circuits from items 1 and 2 in Theorem 2 are within a poly-logarithmic factor of the trivial lower bound of nm. For directed connectivity, the fan-in-2 circuits of smallest depth have depth $O(\log^2 n)$ which is matched by the circuits from Theorem 3. For sorting, it is an open problem to make the circuits $\mathbf{NC}^0$-explicit while maintaining small size.

Because directed connectivity is complete for non-deterministic logarithmic space ($\mathbf{NL}$), Theorem 3 has the following corollary.

Corollary 4. *Every language in* $\mathbf{NL}$ *is computable by an* $\mathbf{NC}^0$*-explicit* $\mathbf{NC}^2$ *circuit.*

Our original motivation for Theorem 2 was to obtain an explicit reduction of non-deterministic time to 3SAT, following Van Melkebeek's version which uses sorting circuits [vM06, §2.3.1], cf. [NEU12]. This was indeed achieved in a prequel to [JMV13]. Shortly afterwards however it was realized that for this reduction to 3SAT it is easier and more efficient to use switching networks instead of sorting, see [JMV13]. Indeed, with switching networks a constant-locality result was obtained whereas, again, it is an open problem to improve Item 1 in Theorem 2 to $\mathbf{NC}^0$-explicitness. Still, the reduction using sorting does not use extra non-determinism, which may turn out to be useful.

Other than this, we do not have a specific application for our results. But the questions appear natural to us.

1.1 Techniques

The proof of our theorems proceed in two stages (in fact the proof of Theorem 3 is more streamlined than described here). First we show how to compute connections by a poly-time algorithm that operates in place: given a gate label g and a bit $b \in \{0,1\}$ specifying one of g's children, the algorithm overwrites g with the child's label using only $O(\log|g|)$ extra space. We note that this is a different notion from low-space computation. In the latter, the input is read-only and each output bit may be computed by reading all of the input. By contrast, we work with in-place algorithms that during the computation overwrite the input with the output. Further, note that the standard definition of log-space uniform circuits

entails label-computing algorithms that use $O(|g|)$ extra space, i.e. *linear* in the length of a label, while we use $O(\log |g|)$ extra space.

In the second stage, we turn in-place computation into the models in Theorems 2 and 3. We now give more details.

Computing connections in place. Our first technical contribution is to define a labeling of bit-length $t + O(\log t) = \log \widetilde{O}(T)$ for comparators in the odd-even mergesort network [Bat68] of size $\widetilde{O}(2^t)$ (and depth t^2) that sorts $T = 2^t$ elements. We then show that given a label one can compute the labels of its children by an efficient algorithm that operates in place. We were not able to obtain this result with the naive matrix labeling where the label is (x, y) for $x \leq \log^2 T, y \leq T$. Instead, our labels include the paths in the recursion trees arising from the algorithm. We show that viewed this way, one can compute children by performing the following operations on labels: comparison, addition/subtraction by 1, bit-shift, and bit-reversal. Each of these operations can be performed in time polynomial in the label size, and in place. We note that there are many other networks that sort T elements in size $\widetilde{O}(T)$. But we have not been able to easily compute children in any other network. To mention one, it is not clear to us how to perform the arithmetic in $\widetilde{O}(T)$-size Shellsort variants, see e.g. [Sed96].

For directed connectivity, a similar labeling (specifically one using the paths in trees arising from the algorithm) can be defined for the standard $\mathbf{NC}^2$ circuit that uses repeated squaring to compute the nth power of an n-vertex graph. The operations needed to compute connections for this labeling are a subset of those needed for the odd-even mergesort network.

These results are sufficient to obtain $\mathbf{AC}^0$-explicit circuits (i.e. ones whose connections are computable in $\mathbf{AC}^0$) for either sorting or directed connectivity, because the operations mentioned above can be computed in that class. But for more restricted classes one needs another idea. Indeed, decision trees require maximum depth to compare two strings, or to add 1 to a binary number.

Making in-place computation local. We employ a general technique called *spreading computation* from [JMV13], which permits a tradeoff between the size and depth of a circuit and the complexity of computing its connections. In [JMV13] this technique is applied to any log-space uniform circuit C to obtain an equivalent circuit C' of size $|C'| = \text{poly}(|C|)$ whose connections can be computed by a local (a.k.a. $\mathbf{NC}^0$) algorithm. In contrast, here we use it to show that any circuit C whose connections can be computed efficiently and in place has an equivalent circuit C' of *quasilinear* size $|C'| = \widetilde{O}(|C|)$ whose connections can be computed by small-depth decision trees.

The main idea of spreading computation is simply to let the gates of C' represent configurations of the algorithm computing children in C. Then computing a child amounts to performing one step of the in-place algorithm, (each bit of) which can be done by a decision tree of depth $O(\log h)$ where h is the label size. Basically, this tree just reads the register variables of the algorithm, and reads/updates the indexed memory bit. This completes the overview of Item 1 in Theorem 2, of Theorem 3, and of Corollary 4.

For Item 2 of Theorem 2, we construct a linear-size log-depth circuit that computes one step of the in-place algorithm. Finally, Item 3 of Theorem 2 requires a technical variant of

the notion of in-place algorithm. Specifically, we break the label of $h := t + O(\log t)$ bits into b blocks each of size h/b, and reserve extra $O(\log h/b)$ bits per block as "control bits" (or state) for that block (and for a fraction of x). Each pointer of the in-place algorithm is then simulated by b pointers, each ranging over a block. We then show that the computation of one step of the algorithm can be done in polynomial time by passing information between control bits of adjacent blocks.

Organization. In §2 we construct the $\mathbf{NC}^0$-explicit circuits for directed connectivity, and prove Theorem 3 and Corollary 4. In §3 we prove our results on the odd-even mergesort network. Finally in §4 we apply the "spreading computation" technique to in-place algorithms, and give the proof of Theorem 2.

2 NL in $\mathbf{NC}^0$-explicit $\mathbf{NC}^2$

In this section we study explicit circuits for the *directed connectivity* problem STCONN. Recall this is the problem of determining, given a directed graph G with designated vertices s and t, whether there is a path from s to t. It is well-known that STCONN is solvable in $\mathbf{NC}^2$ and that it is complete for the class **NL** of non-deterministic, log-space algorithms.

We show that STCONN has $\mathbf{NC}^0$-explicit $\mathbf{NC}^2$ circuits, i.e. polynomial-size $O(\log^2 n)$-depth circuits whose connections are computable with constant locality. In fact, we show that connections in the standard graph-powering circuit (reviewed below) can be computed with this complexity.

Theorem 3 (Succinct, explicit STCONN circuits)**.** *For all $n \in \mathbb{N}$, there is an $\mathbf{NC}^0$-explicit $\mathbf{NC}^2$ circuit $C : \{0,1\}^{n^2} \to \{0,1\}$ deciding directed connectivity in n-vertex graphs.*

Because STCONN is **NL**-complete under a simple reduction, we also obtain the following corollary which is proved afterwards.

Corollary 4. *Every language in **NL** is computable by an $\mathbf{NC}^0$-explicit $\mathbf{NC}^2$ circuit.*

Proof of Theorem 3. Consider the circuit $C' : \{0,1\}^{n^2} \to \{0,1\}^{n^2}$ that, on input the $n \times n$ adjacency matrix M of a directed graph G, outputs the adjacency matrix M' of the graph G^2. (Recall that G^r has an edge (u, v) iff there is a path of length $\leq r$ from u to v in G.) C' has depth $\log n + 1$ and size $\text{poly}(n)$ as shown by the following formula. (We assume that $M_{ii} = 1$ for all $0 \leq i \leq n-1$.)

$$\forall i, j \in \{0, ..., n-1\} : \quad M'_{ij} = \bigvee_{k=0}^{n-1} (M_{ik} \wedge M_{kj}) \tag{1}$$

By concatenating $\log n$ copies of C' we get a polynomial-size circuit C of depth $O(\log^2 n)$ that outputs the adjacency matrix of $G^{2^{\log n}} = G^n$. Thus for designated vertices s and t, the (s,t)-th output bit of C solves STCONN because G^n has an edge (s,t) iff G has a path from s to t. We now define a labeling of C's gates and show how to compute connections between them in $\mathbf{NC}^0$.

A label $\ell = (w, m, h, g) \in \{0,1\}^{O(\log n)}$ has four fields. The 2-bit field w designates the type of the gate which is either OR, AND, or INPUT. For INPUT gates, the next $2\log n$ bits are its index (i.e. (i, j) for some $0 \le i, j \le n-1$) and the remaining bits are 0. For OR and AND gates, the fields m, h, g are used as follows.

- $m \in \{0,1\}^{2\log\log n}$ designates which copy of C' the gate lies in. Recall that C is composed of $\log n$ copies of C' each computing the square of a given graph. To allow m to be incremented in $\mathbf{NC}^0$, we use the following redundant representation which explicitly specifies the carry bits arising from addition. View m as a sequence of pairs

$$((c_{\log\log n}, b_{\log\log n}), \ldots, (c_1, b_1)) \in \{0,1\}^{2\log\log n}.$$

  Then to increment by 1 we simultaneously set $c_1 \leftarrow b_1$, $b_1 \leftarrow b_1 \oplus 1$, and $c_i \leftarrow b_i \wedge c_{i-1}$ and $b_i \leftarrow b_i \oplus c_{i-1}$ for all $i > 1$. To allow an $\mathbf{NC}^0$ circuit to check if m is at its maximum value, our convention is that this maximum is reached when the most-significant bit is 1, which happens after $\log n + \log\log n - 1$ increments. (Thus C technically computes the graph $G^{(n\log n)/2}$, but $G^r = G^n$ for all $r \ge n$ when G has n vertices.)

- $h = (i, j) \in \{0,1\}^{2\log n}$ designates a tree in a copy of C'. Recall that C' has n^2 trees (defined by equation (1)) computing the n^2 bits of an $n \times n$ adjacency matrix.

- $g \in \{0,1\}^{2\log n}$ designates a gate in the tree specified by m and h. For an OR gate, the first $\log n$ bits p contain a path in the OR tree starting from its top gate, and the second $\log n$ bits p' have the form $1^r 0^{(\log n)-r}$ (for $0 \le r \le \log n$) indicating that the path in p has length r. For an AND gate, the first $\log n$ bits are its index, i.e. the value of k in equation (1), and the remaining bits are unused.

Given a label ℓ of a gate in C and a bit b, we can compute the label of the b-th child of ℓ in $\mathbf{NC}^0$ as follows.

Suppose $\ell = (w, m, h, g)$ designates an OR gate in a tree defined by equation (1). By construction it is not at the bottom level of this tree, so first we extend the path encoded in $g = (p, p')$ by one step in the direction corresponding to b, i.e. we set

$$p_i \leftarrow \begin{cases} b & \text{if } p'_i = 0 \wedge p'_{i-1} = 1 \\ p_i & \text{otherwise} \end{cases} \quad \text{and} \quad p'_i \leftarrow \begin{cases} 1 & \text{if } p'_i = 0 \wedge p'_{i-1} = 1 \\ p'_i & \text{otherwise} \end{cases}$$

for each $1 \le i \le \log n$ (for convienience we define $p'_0 = 1$). Note that this is computable in $\mathbf{NC}^0$. If now $p'_{\log n} = 0$ then the child is not at the bottom of the tree, so it is an OR gate and we are done. If instead $p'_{\log n} = 1$ then the child is an AND gate at the bottom of the tree, so we change the type w to AND; no further changes are needed because in this case the new path p is equal to (the binary representation of) the index k of the AND gate.

Now suppose $\ell = (w, m, h, g)$ designates an AND gate. Then $h = (i, j)$ designates a tree in the m-th copy of C' and the first $\log n$ bits of g are the value of k in equation (1). Both of ℓ's children are output gates from the $(m+1)$-th copy of C', unless m is at its maximum value in which case both children are input gates to C. Thus if m is not at its maximum value, we set the type w to OR, increment m, set h to either (i, k) or (k, j) according to b, and set g to all 0. If instead m is at its maximum value, we set the type w to INPUT, and

the index of the relevant input bit is again either (i,k) or (k,j). Note that all operations are computable in $\mathbf{NC}^0$, including those on m as discussed above. □

Finally we show that because STCONN is complete for $\mathbf{NL}$, we can obtain $\mathbf{NC}^0$-explicit $\mathbf{NC}^2$ circuits for all of $\mathbf{NL}$. We use the following lemma from [JMV13] which converts logspace-uniform circuits to $\mathbf{NC}^0$-explicit circuits. ([JMV13, Thm. 5] does not explicitly note the depth blowup, but the proof shows that it is proportional to the running time of the logspace TM.)

Lemma 5 ([JMV13])**.** *Let $f : \{0,1\}^n \to \{0,1\}$ be a function computable by a family of D-explicit polynomial-size circuits $\{C_n\}_n$ for $D =$ TMs running in space $O(\log n)$ and time $t(n) \leq n^{O(1)}$. Then f is also computable by a family of polynomial-size $\mathbf{NC}^0$-explicit circuits $\{C'_n\}_n$ such that $\mathsf{Depth}(C'_n) \leq O(t(n)) \cdot \mathsf{Depth}(C_n)$.*

The specific TM model used in Lemma 5 is the following. There is a single tape of length $O(\log n)$ that initially contains an $O(\log n)$-bit gate label, n written in binary, and a bit indicating which child label to compute. The tape head moves sequentially (i.e. one cell left or right in each step), and at the end the tape contains the type and label of the child gate. What we require in the following proof is that such a TM can, in time $O(\log^2 n)$, check whether two $O(\log n)$-bit configurations of another non-deterministic logspace TM are adjacent. This can be done in a straightforward manner (details omitted). In fact the time for this step can be reduced to $O(\log n)$ by interleaving the two configurations on the tape, though this does not improve the overall result.

Proof of Corollary 4. Let $f \in \mathbf{NL}$ be a language decided by a log-space NTM M. Recall that the standard reduction from f to STCONN produces, on input $x \in \{0,1\}^n$, the $n^{O(1)} \times n^{O(1)}$ adjacency matrix of $M(x)$'s configuration graph, i.e. where the (i,j)-th bit is 1 iff the j-th configuration is reachable from the i-th configuration in one step on input x.

Let C be the $\mathbf{NC}^0$-explicit $\mathbf{NC}^2$ circuit from Theorem 3 that decides STCONN. Next we show that for every pair of configurations i, j of M, there is an $\mathbf{NC}^0$-explicit $\mathbf{NC}^2$ circuit $C_{ij} : \{0,1\}^n \to \{0,1\}$ such that

(a) $C_{ij}(x) = 1$ iff $M(x)$ can reach configuration j in one step from configuration i, and

(b) the label of C_{ij}'s output gate can be computed in $\mathbf{NC}^0$ from i, j.

Given such C_{ij}, we modify C by replacing its (i,j)-th input gate with the output gate of C_{ij}. The resulting $\mathbf{NC}^2$ circuit (on input x) decides $f(x)$ due to (a) and the correctness of the reduction. Further, it is $\mathbf{NC}^0$-explicit because C and all C_{ij} are, and we can go from the (i,j)-th input gate label of C to the output gate label of C_{ij} in $\mathbf{NC}^0$ by (b). We now construct the circuits C_{ij}.

A configuration of M contains the $(\log n)$-bit input tape head position (i.e. an index into x) and $O(\log n)$ bits representing the work tape (including the state and work tape head). For configurations i and j, let p denote i's input tape head position. Then C_{ij} is an OR of two ANDs, where for $b \in \{0,1\}$ the b-th AND checks that $x_p = b$ and that j follows from i assuming $x_p = b$. The latter check is completely determined by i, j, and b, so the inputs to each AND are the gate outputting x_p (possibly negated) and a constant gate. The

label for the gate outputting x_p can be easily computed because wlog p is stored in i at a fixed location. The label for the constant gate can be computed by a TM of the type in Lemma 5 running in time $O(\log^2 n)$, because as noted above such TMs can check whether the $O(\log n)$-bit configurations i and j are adjacent (given the input bit read by i). Thus we can take C_{ij} to be a (time $O(\log^2 n)$)-explicit $\mathbf{NC}^0$ circuit, and by applying Lemma 5 we can take it to be an $\mathbf{NC}^0$-explicit $\mathbf{NC}^2$ circuit. □

We conclude this section with a remark about "invalid" gate labels in the above $\mathbf{NC}^0$-explicit circuits for STCONN and **NL**. Following [JMV13], any string that can be obtained by starting with the output gate label and repeatedly applying the connection-computing algorithm is a valid label, and all other strings of the same length are invalid labels. Such invalid labels are problematic if they contain a cycle with an odd number of NOT gates, as then the computation is not well-defined. However the circuits constructed above are monotonic, i.e. there are *no* NOT gates except for those that negate a bit of the input x in the proof of Corollary 4, and thus no such cycles can exist.

3 Explicit odd-even mergesort

In this section we describe our encoding of the odd-even mergesort network, and then we prove that given an index to a comparator and a bit $b \in \{0,1\}$, one can compute the child b of the comparator with an algorithm that operates in place and runs in polynomial time in the length of the index. Recall that an in-place algorithm for a function that maps say $\{0,1\}^n \to \{0,1\}^n$ is one that is computable using a constant number of variables that each hold $O(\log n)$ bits, and no additional space (using the standard RAM arithmetic and indexing).

First we recall the odd-even mergesort algorithm, which is just like Mergesort except that the (input-dependent) merge subroutine is replaced with a subroutine whose comparisons do not depend on the input. In the rest of this section we abbreviate Odd-Even by OE.

Algorithm 2, OE-MERGE, merges the two already sorted halves of the sequence $A = [a_0, a_1, \ldots, a_{2^t-1}]$, resulting in a sorted output sequence. It uses an operation CMP-EX(A, i, j) that compares values at indices $i < j$ of the sequence A and exchanges them if and only if $a_i > a_j$. (CMP-EX is short for Compare-Exchange.) Algorithm 1, OE-MERGESORT, uses OE-MERGE as a subroutine. The proof of correctness is standard and short; for an exposition see e.g. [NEU12].

Algorithm 1: OE-MERGESORT(A)

input : Sequence $A = [a_0, \ldots, a_{2^t-1}]$.
output: The sequence A in sorted order.

```
1 if t ≥ 1
2 |  OE-MERGESORT([a_0, a_1, ..., a_{2^{t-1}-1}])        /* first half */
3 |  OE-MERGESORT([a_{2^{t-1}}, a_{2^{t-1}+1} ..., a_{2^t-1}])  /* second half */
4 |  OE-MERGE(A)
```

Algorithm 2: OE-MERGE(A)

input : Sequence $A = [a_0, \ldots, a_{2^t-1}]$ of length $2^t \geq 2$, such that $[a_0, a_1, \ldots, a_{2^{t-1}-1}]$ and $[a_{2^{t-1}}, a_{2^{t-1}+1}, \ldots, a_{2^t-1}]$ are each sorted.

output: The sequence A in sorted order.

```
if |A| = 2
    CMP-EX(A, 0, 1)
else
    OE-MERGE([a_0, a_2, ..., a_{2^t-2}])   /* the even subsequence */
    OE-MERGE([a_1, a_3, ..., a_{2^t-1}])   /* the odd subsequence */
    for i ∈ {1, 3, 5, ..., 2^t - 3}
        CMP-EX(A, i, i + 1)
```

The mergesort label format. Consider Algorithm 1 on an input of length $T = 2^t$. We can see that it is a complete depth-$(t-1)$ binary tree of OE-MERGE operations. An OE-MERGE operation at depth $i = t - k$ in this tree ($0 \leq i \leq t-1$) merges two sorted lists, each of size 2^{k-1}, into a sorted list of size 2^k. The inputs of an OE-MERGE operation in this tree are the outputs of its two children, except for the leaves which take their input from the initial list of configurations (i.e. the algorithm's input). So, the first part of a comparator's label is a binary string p of length $i \leq t-1$, specifying a path through (and thus a node of) this tree.

Consider some OE-MERGE operation B at depth $i = t - k$, so with 2^k outputs; we refer to such B as a "2^k-merge-block". As can be seen from Algorithm 2, B is composed of two parallel 2^{k-1}-merge-blocks B_{even} and B_{odd} that respectively merge B's even- and odd-indexed inputs, followed by $2^{k-1} - 1$ parallel comparators that produce the final sorted list. B_{even} and B_{odd} are themselves recursively constructed in this way, and so we can view B itself as a complete depth-$(k-1)$ binary tree, where a node at depth ℓ ($0 \leq \ell \leq k-1$) contains the $2^{k-\ell-1}$ comparators at the output of a $2^{k-\ell}$-merge-block inside B. For example, the root contains the $2^{k-1} - 1$ comparators at B's output, its left child contains the $2^{k-2} - 1$ comparators at B_{even}'s output, and so on; in particular, the leaves (i.e. when $|A| = 2$ in Algorithm 2) are the comparators sitting at the input of B. Thus, we can specify a comparator within B with the following two items: a path p' of length $\ell \leq k-1$ through this binary tree, and a number $1 \leq m \leq 2^{k-\ell-1} - 1$ identifying a comparator within the corresponding node. (When $\ell = k-1$ there is no such m, but in this case p' uniquely specifies the comparator.)

We want a label to encode p, p', and m, where recall p specifies a path to a merge block B, p' specifies a path within B to an internal merge block B', and m specifies a comparator at the output of B'. Some care is required to ensure that each label has size $t + o(t)$. For example, p and p' can have length up to $t-1$, so we cannot afford to store them separately. We exploit the following tradeoff: when p is a path of length i, then p' is a path of length $\ell \leq t - i - 1$, and the number of comparators at this level is $2^{t-i-\ell-1} - 1$. Thus for fixed i and ℓ one can encode a label with $i + \ell + (t - i - \ell - 1) = t - 1$ bits. We also separately encode i and ℓ, for a total encoding length of $t + O(\log t)$ bits.

Definition 6 (Odd-even mergesort labels)**.** Our encoding of the odd-even mergesort network

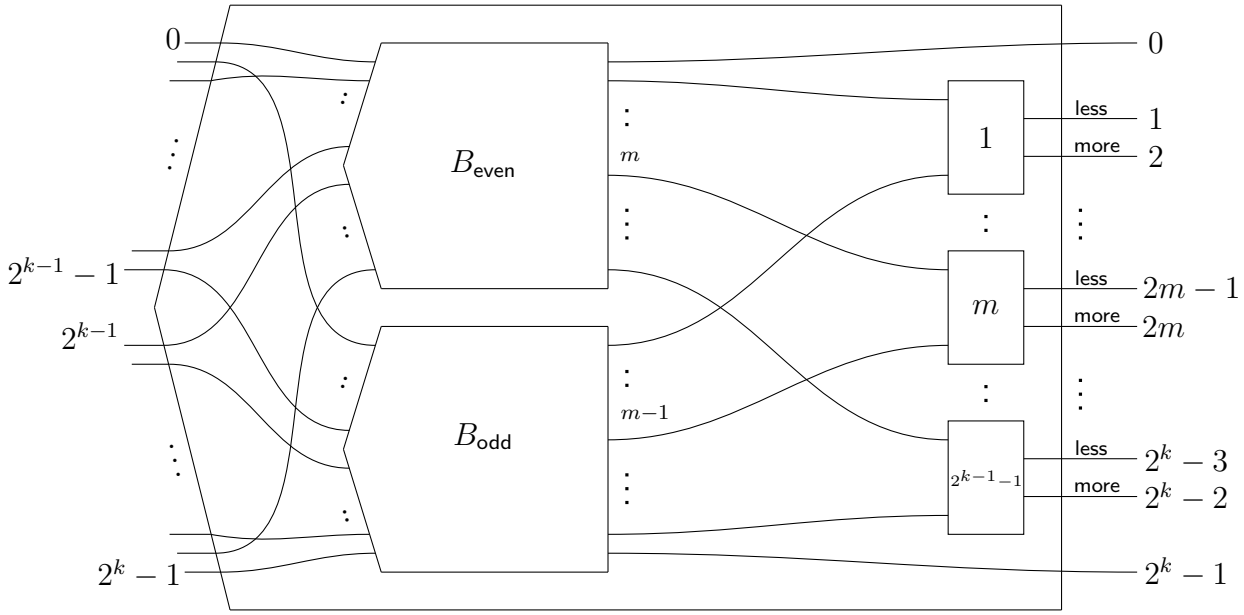

Figure 1: The recursive structure of a 2^k-merge-block B and its connections.

on 2^t inputs has labels of $t-1+2\log t$ bits, as follows. First, we have a string $z \in \{0,1\}^{t-1}$ that encodes the concatenation of p, p', and the binary representation of m. Second, we have two indices $0 \le i, i' \le t-1$ that specify the "breakpoints" in the concatenation. More formally, given $(z, i, i') \in \{0,1\}^{t-1} \times \{0,\ldots,t-1\} \times \{0,\ldots,t-1\}$, we parse the path $p := z_1 \cdots z_i$, the path $p' := z_{i+1} \cdots z_{i'}$, and $m :=$ (the number encoded in binary by) $z_{i'+1} \cdots z_{t-1}$.

Note that the length ℓ of p' is equal to $i' - i$; storing i' rather than ℓ is more natural for the following algorithm.

We observe that every triple $(z, i, i') \in \{0,1\}^{t-1} \times \{0,\ldots,t-1\} \times \{0,\ldots,t-1\}$ is a valid label of a comparator, except for the following two cases: (1) if $i > i'$; or (2) if $z_{i'+1} = \cdots = z_{t-1} = 0$, because we number comparators starting at 1.

Computing on the mergesort labels. We now describe an algorithm for computing the labels of a comparator's two children, given its label (z, i, i'). The complete algorithm is in Algorithm 3. We will describe later why this algorithm may be executed in place; but first we show that the topology induced by Algorithm 3 matches the circuit specified by Algorithms 1 and 2.

For a given label, we need to compute the labels of its two children, i.e. the two comparators from which its inputs come. Each comparator has two inputs and two outputs; the order of the inputs is unimportant because the comparator sorts them, but the order of the outputs obviously matters. Thus for each child, we must specify both the comparator's label and one of its outputs. We refer to the two outputs of each comparator as less and more.

In describing the algorithm, it will be helpful to refer to the output wires of a given merge block. For a merge block B with 2^k output wires, we number these wires as $0,\ldots,2^k-1$ (in contrast to the comparators which are numbered starting from 1). Referring to Figure

Algorithm 3: Algorithm for computing child labels in the Odd-Even Mergesort circuit

input : $(z, i, i') \in \{0,1\}^{t-1} \times \{0,\ldots,t-1\} \times \{0,\ldots,t-1\}$
output: children $\in \{0,1\}^{t-1} \times \{0,\ldots,t-1\} \times \{0,\ldots,t-1\} \times \{\mathsf{less}, \mathsf{more}\}$

if $i' < t-2$ `/* children in same merge block, not both at the bottom */`
  if $z_{i'+1} = \cdots = z_{t-2} = 0$ **and** $z_{t-1} = 1$ `/* m = 1 (first comparator) */`
    $\mathsf{children} = \begin{array}{ll} (z_1 \cdots z_{i'} \circ 0 \circ z_{i'+2} \cdots z_{t-1}, & i, \quad i'+1, \quad \mathsf{less}) \\ (z_1 \cdots z_{i'} \circ 1 \circ 0^{t-i'-2}, & i, \quad t-1, \quad \mathsf{less}) \end{array}$
  else if $z_{i'+1} = \cdots = z_{t-1} = 1$ `/* m = 2^{t-i'-1} - 1 (last comparator) */`
    $\mathsf{children} = \begin{array}{ll} (z_1 \cdots z_{i'} \circ 0 \circ 1^{t-i'-2}, & i, \quad t-1, \quad \mathsf{more}) \\ (z_1 \cdots z_{i'} \circ 1 \circ z_{i'+2} \cdots z_{t-1}, & i, \quad i'+1, \quad \mathsf{more}) \end{array}$
  else
    if $z_{t-1} = 0$ `/* m is even */`
      Let $x :=$ $(t-i'-2)$-bit representation of $m/2$
      $\mathsf{children} = \begin{array}{ll} (z_1 \cdots z_{i'} \circ 0 \circ x, & i, \quad i'+1, \quad \mathsf{more}) \\ (z_1 \cdots z_{i'} \circ 1 \circ x, & i, \quad i'+1, \quad \mathsf{less}) \end{array}$
    else if $z_{t-1} = 1$ `/* m is odd */`
      Let $x :=$ $(t-i'-2)$-bit representation of $(m+1)/2$
      Let $x' :=$ $(t-i'-2)$-bit representation of $(m-1)/2$
      $\mathsf{children} = \begin{array}{ll} (z_1 \cdots z_{i'} \circ 0 \circ x, & i, \quad i'+1, \quad \mathsf{less}) \\ (z_1 \cdots z_{i'} \circ 1 \circ x', & i, \quad i'+1, \quad \mathsf{more}) \end{array}$
else if $i' = t-2$ `/* children at bottom level of same merge block */`
  $\mathsf{children} = \begin{array}{ll} (z_1 \cdots z_{t-2} \circ 0, & i, \quad t-1, \quad \mathsf{more}) \\ (z_1 \cdots z_{t-2} \circ 1, & i, \quad t-1, \quad \mathsf{less}) \end{array}$
else if $i' = t-1$ `/* children not in same merge block */`
  if $z_{i+1} = \cdots = z_{t-1} = 0$ `/* bottom-left-most comparator */`
    $\mathsf{children} = \begin{array}{ll} (z_1 \cdots z_i \circ 0 \circ 0^{t-i-2}, & i+1, \quad t-1, \quad \mathsf{less}) \\ (z_1 \cdots z_i \circ 1 \circ 0^{t-i-2}, & i+1, \quad t-1, \quad \mathsf{less}) \end{array}$
  else if $z_{i+1} = \cdots = z_{t-1} = 1$ `/* bottom-right-most comparator */`
    $\mathsf{children} = \begin{array}{ll} (z_1 \cdots z_i \circ 0 \circ 1^{t-i-2}, & i+1, \quad t-1, \quad \mathsf{more}) \\ (z_1 \cdots z_i \circ 1 \circ 1^{t-i-2}, & i+1, \quad t-1, \quad \mathsf{more}) \end{array}$
  else
    Let $m :=$ integer represented by $z_{t-1} z_{t-2} \cdots z_{i+1}$
    if $z_{i+1} = 0$ `/* m is even */`
      Let $x :=$ $(t-i-2)$-bit representation of $m/2$
      $\mathsf{children} = \begin{array}{ll} (z_1 \cdots z_i \circ 0 \circ x, & i+1, \quad i+1, \quad \mathsf{more}) \\ (z_1 \cdots z_i \circ 1 \circ x, & i+1, \quad i+1, \quad \mathsf{more}) \end{array}$
    else if $z_{i+1} = 1$ `/* m is odd */`
      Let $x :=$ $(t-i-2)$-bit representation of $(m+1)/2$
      $\mathsf{children} = \begin{array}{ll} (z_1 \cdots z_i \circ 0 \circ x, & i+1, \quad i+1, \quad \mathsf{less}) \\ (z_1 \cdots z_i \circ 1 \circ x, & i+1, \quad i+1, \quad \mathsf{less}) \end{array}$

1, one sees that the mth wire when m is even is the more output of comparator $m/2$ at the top level of B, and when m is odd it is the less output of comparator $(m+1)/2$. The only exceptions are the first and last wires, which bypass the comparators at B's top level. For these, the first wire is the less output of B's bottom-left-most comparator (i.e. the one with path $p' = 0^{k-1}$ in B) and the last wire is the more output of B's bottom-right-most comparator (the one with path $p' = 1^{k-1}$).

Note that there are two types of connections: those within a merge block, i.e. within a group of comparators arising from a call to OE-MERGE by OE-MERGESORT, and those between merge blocks. In the former the children have the same path p as their parent, while in the latter the children's path p is one bit longer than their parent's (one child has $p \circ 0$, and the other has $p \circ 1$).

We begin with the connections within a merge block, corresponding to lines 1-15 in Algorithm 3. Let (z, i, i') specify a comparator at the output of some 2^k-merge-block B (where $k = (t-1) - i' > 0$). B contains $2^{k-1} - 1$ comparators at its output level, and two recursively constructed 2^{k-1}-merge-blocks B_{even} and B_{odd} below them (refer to Figure 1). B's top-level comparators are connected to B_{even} and B_{odd} via the following rule: for $1 < m < 2^{k-1} - 1$, the mth comparator takes as input the mth output wire from B_{even} and the $(m-1)$th output from B_{odd}. Thus, using the correspondence between output wires and comparators mentioned above, the children of the mth comparator are computed as follows (lines 7-13 of Algorithm 3). If m is even (line 7), then the children are the $(m/2)$-th output comparators of B_{even} and B_{odd}. If m is odd (line 10), then the children are the $((m+1)/2)$-th output comparator of B_{even} and the $((m-1)/2)$-th output comparator of B_{odd}.

This accounts for all of B's top-level comparators except the first ($m = 1$) and last ($m = 2^{k-1} - 1$). For each of these, one of the input wires is either the first or last output of B_{even} or B_{odd}, and as mentioned above these wires come from the bottom-level comparators. For the first comparator (lines 2-3 of Algorithm 3), one child comes from the first top-level comparator of B_{even} and the other from the bottom-left-most comparator of B_{odd}. For the last comparator (lines 4-5), one child comes from the last top-level comparator of B_{odd} and the other from the bottom-right-most comparator of B_{even}.

The only remaining special case for connections within a merge block is when both children are at the bottom level of B (i.e. (z, i, i') is one level above the bottom), which is indicated by $i' = t - 2$ (line 14 of Algorithm 3). In this case the children are uniquely specified by their paths p, p' and have no number m; they are obtained by setting $i' = t - 1$ and setting the last bit of z to be 0 or 1.

We now describe the connections *between* merge blocks, i.e. how to compute the children of the comparators at the bottom level of a merge block B, corresponding to lines 16-28 in Algorithm 3.

Let B denote a 2^k-merge-block at some depth $i = t - k$, and let B_0 and B_1 denote its two children at depth $i + 1$. Let $(z, i, i' = t - 1)$ be the label of a comparator at the bottom level of B, which thus has one child in B_0 and one in B_1. The unique aspect of these connections is the following. Because (z, i, i') is at B's bottom level, it has a path p of length i and a path p' of length $t - i - 1$, but no number m. In contrast, comparators at the top level of B_0 and B_1 have a path p and a number m, but no path p'. Thus, we must convert the parent's path p' into the number m for each of its children.

The way that this is done is reminiscent of the fast Fourier transform or butterfly network. Consider the 2^k wires that are input to B, and label them $0, \ldots, 2^k - 1$ (wires $0, \ldots, 2^{k-1} - 1$ are output by B_0 and wires $2^{k-1}, \ldots, 2^k - 1$ are output by B_1). We need to determine which comparator at B's bottom level each wire is input to. From lines 4-5 of Algorithm 2, we can see that the wires are first partitioned into the even- and odd-indexed sets, then the sets are recursively partioned again, and so on until each set in the partition contains two wires which are then both input to a single comparator. Now view B's 2^{k-1} bottom-level comparators as the leaves of a complete depth-$(k-1)$ binary tree; indeed, in our label format this corresponds to the path p'. For a given input wire labeled by a k-bit binary number, we find its comparator by routing it on this tree, reading its label from right to left and ignoring the leftmost bit. This gives the correct comparator because reading the labels from right to left corresponds to recursively partitioning the wires into the even- and odd-indexed sets, and ignoring the last bit corresponds to stopping when the sets have size two.

Now notice that we can do this in the other direction as well: given the path p' labeling a comparator at the bottom level of B, the indices of the two wires that it inputs have binary representations $0 \circ \mathsf{reverse}(p')$ and $1 \circ \mathsf{reverse}(p')$. In other words, its input wires are the mth output wires of B_0 and B_1, where $m := \mathsf{reverse}(p')$. Finally by the discussion above mapping output wires to output comparators, we can compute the labels of (z, i, i')'s children from $\mathsf{reverse}(p') = z_{t-1}z_{t-2}\cdots z_{i+1}$ using similar techniques as in the previous cases (and with similar special cases corresponding to the first and last output wires of B_0 and B_1, handled by lines 17-20 of Algorithm 3).

We have finished the proof of the following theorem.

Theorem 7. *Algorithms 3 and 1 induce the same sorting network.*

We now observe that this algorithm can be extended to also compute connections within each comparator, and that the entire computation can be performed by an in-place algorithm.

Theorem 8. *For every $m, n \in \mathbb{N}$, there is a sorting circuit $C : (\{0,1\}^m)^n \to (\{0,1\}^m)^n$ of size $O(mn \log^2 n)$ whose gate connections are computable in place. That is, there is a $(\log |C|)$-bit labeling of C's gates and a polynomial-time in-place algorithm L such that for every gate label g and bit $b \in \{0,1\}$, $L(g, b)$ outputs the bth child of g.*

Proof. Algorithm 3 is easily seen to be computable by an in-place algorithm. The operations used are incrementing/decrementing a binary number by 1, dividing an even binary number by 2 (which is just a shift), and reversing a binary string, all of which can be done in place.

A comparator on two m-bit numbers $x, y \in \{0,1\}^m$ can be implemented by a circuit of size $O(m)$. Indeed, the crux is in determining whether $x > y$ which can be represented by

$$t \;:=\; \bigvee_{i=1}^{m} \left((x_i > y_i) \;\wedge \bigwedge_{j=i+1}^{m} (x_j = y_j) \right).$$

Then the ith bit of the comparator's more output is given by $(x_i \wedge t) \vee (y_i \wedge \neg t)$, while the ith bit of its less output is given by $(x_i \wedge \neg t) \vee (y_i \wedge t)$.

The regularity of these formulas permits an intuitive $O(\log m)$-bit labeling for which connections can be computed by an in-place algorithm; we omit the details. Concatenating this labeling with the odd-even mergesort labeling above gives the required labeling of C. $\square$

4 Spreading computation

In this section we show that the circuit C from Theorem 8 has an equivalent circuit C' of size $\widetilde{O}(|C|)$ whose gate connections can be computed in more restricted computational models, namely log-depth decision trees and log-depth linear-size circuits. This follows from a more general result that any circuit whose connections are computable in place and in polynomial time has an equivalent circuit whose connections are computable in these restricted models.

The idea is to use the *spreading computation* technique of [JMV13], and label the gates of C' by configurations of the in-place algorithm L. Computing connections in C' then corresponds to computing a single step of L, which can be done with more limited resources than are required for an entire run of L. In practice, this corresponds to introducing a chain of "copy" gates between each pair of connected gates $(g_{\mathsf{parent}}, g_{\mathsf{child}})$ in C; the labels of this chain encode L's computation on input g_{parent} with output g_{child}. The increase in size and depth is proportional to L's running time; since the latter is $\log^{O(1)}|C|$, we have $|C'| = \widetilde{O}(|C|)$.

Lemma 9 (In-place $\to$ small depth)**.** *Let L be a polynomial-time in-place algorithm that computes connections between $(h = \log|C|)$-bit labels of a fan-in-2 circuit C. Then there is an equivalent D-explicit circuit C' of size $|C'| = \widetilde{O}(|C|)$ for*

1. *D = decision trees of depth $O(\log h)$, or*
2. *D = circuits of size $O(h)$ and depth $O(\log h)$.*

Proof. The proof follows the ideas of [JMV13, Thm. 5]. The main difference is that they start from a log-space algorithm, while we start from an in-place algorithm.

We model L as follows. L's input is the label $g \in \{0,1\}^h$ and child-selection bit b. L uses a constant number of $(\log h)$-bit variables $p_1, \ldots, p_k$. At each step, L reads its variables and the bit g_{p_1}, and then updates g_{p_1} and each of the variables. An $(h + O(\log h))$-bit configuration of L (a.k.a. a label in C') contains the following items: the current timestep ($O(\log h)$ bits), the value of L's variables ($O(\log h)$ bits), the child-selection bit b (1 bit), and the current string g (h bits).

D computes as described in [JMV13, Thm. 5], namely by performing one step of L from the configuration it is given as input. It remains to argue that D can be computed by decision trees of depth $O(\log h)$ or by circuits of size $O(h)$ and depth $O(\log h)$. For decision trees this is immediate, because the timestep, the variables, the bit b, and the relevant bit of g can all be read in depth $O(\log h)$.

For circuits, we construct $C_h : \{0,1\}^{\log h} \to \{0,1\}^h$ of size $O(h)$ and depth $O(\log h)$ such that $C_h(i) = 2^i$. Indeed, with such a circuit we can fetch a bit of g indexed by $p \le h$, by computing the bit-wise AND of $C_h(p)$ and g, and then computing OR. We can also update the variables which are short and hence easy to compute with, and finally we can update the label bits again by using the circuit C_h (we can XOR its output with the label).

C_h may be constructed from $C_{h/2}$ as follows: parse the input $i = (i_0, i') \in \{0,1\} \times \{0,1\}^{\log h - 1}$; then compute $z_L \in \{0,1\}^{h/2}$ by ANDing i_0 to every bit of $C_{h/2}(i')$, compute $z_R \in \{0,1\}^{h/2}$ by ANDing $(i_0 \oplus 1)$ to every bit of $C_{h/2}(i')$, and output $z_L \circ z_R$. The correctness follows because C_h shifts $C_{h/2}(i') = 2^{i'}$ by $h/2$ bits, i.e. multiplies by $2^{h/2}$, iff the most significant bit of i is 1. The size and depth bounds follow by an inductive argument. □

Combining Theorem 8 and Lemma 9 proves parts 1-2 of Theorem 2.

4.1 A locality vs. size tradeoff

In this section we show a tradeoff between the size and the locality of an in-place circuit-labeling algorithm, proving part 3 of Theorem 2. To this end, we first give a formal definition of in-place circuit-labeling algorithms which is essentially the same formalization used in the proof of Lemma 9. Next, using a technical variant of the definition, we formalize and prove the trade-off.

Definition 10. A *label-processing Turing machine (LTM) L* for a circuit $C : \{0,1\}^n \to \{0,1\}$ is a single-tape Turing machine defined as follows. The tape is read/write and initially contains L's input, a string g of length $h = \log |C|$ which is the label of a gate in C. L has an additional bit c that specifies which of g's children it should compute. At the end of the computation, the label of the designated child gate is written on the tape.

L's control consists of an $O(1)$-bit state and a constant number of pointers $p_1, p_2, \ldots, p_k$ that each have size $\log h$. At each step in the computation L reads the state, all the pointers, and the bit g_{p_1} and then updates the state, g_{p_1}, and the pointers (using standard arithmetic operators).

It can be shown that the in-place circuit-labeling Algorithm 2 is computable by an LTM.

Observe that in order to compute each step of an LTM L given its current configuration, one needs decision trees of depth $\Omega(\log h)$ (e.g. to read the pointers p_1). We now define a modified version of an LTM, called a *block-LTM*, that allows each step to be computed by decision trees of smaller depth, even as small as $O(1)$. This machine is parameterized by $b \in \mathbb{N}$ which allows for a tradeoff between configuration size and decision tree depth.

Definition 11. A *b-block-LTM (b-LTM)* L for a fan-in-2 circuit $C : \{0,1\}^n \to \{0,1\}$ is an LTM with the following modifications. The tape is divided into b blocks of h/b bits. L's control is also divided into b blocks; each block of the control consists of an $O(1)$-bit state and a constant number of pointers $p_1, p_2, \ldots, p_k$ that each have size $\log(h/b)$. Let $[g]^i$ denote the ith block of the tape. At each step in the computation, for each $i \leq b$ in parallel, L updates block i as follows. It first reads control blocks $i-1$, i, and $i+1$ and the bit $[g]^i_{p_1}$ where p_1 is the first pointer in control block i, and it then updates $[g]^i_{p_1}$ and control block i.

We now show that b-LTMs are sufficiently powerful to compute our main algorithm.

Lemma 12. *For every $1 \leq b \leq h$, Algorithm 2 can be computed by a b-LTM.*

Proof. The high-level idea is that the control blocks of a b-LTM can "pass" the state of an LTM between themselves, and thus simulate the LTM's computation. The main technical difficulty in the construction is to simulate the use of a $(\log h)$-bit pointer by a sequence of b $(\log(h/b))$-bit pointers. We now describe how such a set of b pointers can be used in a distributed fashion to specify one of the LTM's pointers.

To start, we recall the operations using $(\log h)$-bit pointers that are needed in the algorithm from Section 3, namely: (1) incrementing or decrementing a pointer by 1, (2) setting

a pointer to a fixed value, (3) reading the bit of the tape specified by a pointer, and (4) reading a string of bits from the tape into a pointer.

We store a $(\log h)$-bit pointer $1 \leq p \leq h$ in a sequence $p_1, \ldots, p_b$ of $(\log(h/b) + 1)$-bit pointers, one per control block, as follows. The first bit of each p_i is an "in-range" flag which is set to 1 iff $(i-1) \cdot h/b \leq p < i \cdot h/b$. In this case the remaining bits store the binary representation of $p - (i-1) \cdot h/b$; otherwise these bits are ignored. (Note that each control block may have additional pointers that are not used in this way, e.g. to represent a marker on some cell in that portion of the tape.)

Given this representation of $(\log h)$-bit pointers, operations (1)-(4) in the above list may be performed by "passing state" between adjacent control blocks. For example, Operation (4) is performed as follows. First, a "distributed pointer" is initialized to 0, and the portion of the tape containing the pointer to be read is copied bit by bit to a new location that is designated for this purpose. (As the pointer has length $\log h$, we can afford this extra space in a configuration.) Next, the copy is repeatedly decremented by 1, and each time the distributed pointer is incremented by 1, stopping when the copy is at 0 (at which point the distributed pointer contains the desired value).

Using these techniques for simulating pointer operations, the rest of the LTM simulation may be carried out. □

We finally prove the main result of this subsection, which in combination with Lemma 12 gives part 3 of Theorem 2.

Theorem 13. *Let L be a b-LTM that computes connections between $(h = \log |C|)$-bit labels of a fan-in-2 circuit C. Then there is an equivalent D-explicit circuit C' of size $|C'| = 2^{h + b \cdot O(\log(h/b))} = |C| \cdot ((\log |C|)/b)^{O(b)}$ for $D =$ decision trees of depth $O(\log(h/b))$.*

Proof. Observe that the total space needed for the control of a b-LTM is $b \cdot O(\log(h/b))$ bits, and thus a configuration of a b-LTM requires $h + b \cdot O(\log(h/b))$ bits. Further observe that given as input a configuration of a b-LTM, a decision tree of depth $O(\log(h/b))$ can compute each bit in the configuration that follows in one step: each bit depends on ≤ 3 control blocks and ≤ 1 bit of each tape.

The only caveat in adapting Lemma 9 to this setting is in the use of timesteps in the configurations. As in [JMV13, Thm. 5], the type of a gate in C' is *copy* iff the ($O(\log h)$-bit) timestep is less than its maximum value (otherwise the type matches the corresponding gate in C). There are two issues when using decision trees of depth $O(\log(h/b))$: how to increment a timestep by 1, and how to check if the timestep is maximal. To handle both of these we use the redundant representation of timesteps employed in [JMV13] and in the proof of Theorem 3 above, which allows these operations to be computed in depth $O(1)$ (see the discussion there for details). □

References

[Ajt83] Miklós Ajtai. $\Sigma^1[1]$-formulae on finite structures. *Ann. Pure Appl. Logic*, 24(1):1–48, 1983.

[Ajt93] Miklós Ajtai. Approximate counting with uniform constant-depth circuits. In *Advances in computational complexity theory*, pages 1–20. Amer. Math. Soc., Providence, RI, 1993.

[All89] Eric Allender. P-uniform circuit complexity. *J. of the ACM*, 36(4):912–928, 1989.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[BIS90] David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *J. of Computer and System Sciences*, 41(3):274–306, 1990.

[BKLM12] Christoph Behle, Andreas Krebs, Klaus-Jörn Lange, and Pierre McKenzie. The lower reaches of circuit uniformity. In *Symp. on Math. Foundations of Computer Science (MFCS)*, pages 590–602, 2012.

[JMV13] Hamid Jahanjou, Eric Miles, and Emanuele Viola. Local reductions. 2013.

[NEU12] NEU. From RAM to SAT. Available at http://www.ccs.neu.edu/home/viola/, 2012.

[Ruz81] Walter L. Ruzzo. On uniform circuit complexity. *J. of Computer and System Sciences*, 22(3):365–383, 1981.

[Sed96] Robert Sedgewick. Analysis of shellsort and related algorithms. In *European Symposium on Algorithms (ESA)*, pages 1–11, 1996.

[Vio09] Emanuele Viola. On approximate majority and probabilistic time. *Computational Complexity*, 18(3):337–375, 2009.

[vM06] Dieter van Melkebeek. A survey of lower bounds for satisfiability and related problems. *Foundations and Trends in Theoretical Computer Science*, 2(3):197–303, 2006.

[Vol99] Heribert Vollmer. *Introduction to circuit complexity*. Springer-Verlag, Berlin, 1999.