Shuffles and Circuits
(On Lower Bounds for Modern Parallel Computation)[*]

Tim Roughgarden[†] Sergei Vassilvitskii[‡] Joshua R. Wang[§]

August 16, 2017

Abstract

The goal of this paper is to identify fundamental limitations on how efficiently algorithms implemented on platforms such as MapReduce and Hadoop can compute the central problems in the motivating application domains, such as graph connectivity problems.

We introduce an abstract model of massively parallel computation, where essentially the only restrictions are that the "fan-in" of each machine is limited to s bits, where s is smaller than the input size n, and that computation proceeds in synchronized rounds, with no communication between different machines within a round. Lower bounds on the round complexity of a problem in this model apply to every computing platform that shares the most basic design principles of MapReduce-type systems.

We prove that computations in our model that use few rounds can be represented as low-degree polynomials over the reals. This connection allows us to translate a lower bound on the (approximate) polynomial degree of a Boolean function to a lower bound on the round complexity of every (randomized) massively parallel computation of that function. These lower bounds apply even in the "unbounded width" version of our model, where the number of machines can be arbitrarily large. As one example of our general results, computing any non-trivial monotone graph property — such as connectivity — requires a super-constant number of rounds when every machine receives only a sub-polynomial (in n) number of input bits s.

Finally, we prove that, in two senses, our lower bounds are the best one could hope for. For the unbounded-width model, we prove a matching upper bound. Restricting to a polynomial number of machines, we show that asymptotically better lower bounds would separate P from NC^1.

1 Introduction

The past decade has seen a resurgence of parallel computation, and there is now an impressive array of frameworks built for working with large datasets: MapReduce, Hadoop, Pregel, Giraph, Spark, and so on. The goal of this paper is to identify fundamental limitations on how efficiently these frameworks can compute the central problems in the motivating application domains, such as graph connectivity.

Modern architectures for massively parallel computation share a number of common attributes. First, the data, or the input to the computation, is partitioned arbitrarily across all of the nodes participating in

[*]A preliminary version of this paper appeared in the 28th ACM Symposium on Parallelism in Algorithms and Architectures, July 2016.

[†]Stanford University. Supported in part by NSF Grant CCF-1524062.

[‡]Google, Inc.

[§]Stanford University. Supported in part by a Stanford Graduate Fellowship and NSF Grant CCF-1524062.

the computation. Second, computation proceeds in synchronous rounds. In every round, each machine looks at the local data available and performs some amount of computation *without communicating with other machines*. After the computation is done, a communication round begins, where each machine can send messages to other machines. Importantly, there are no restrictions on the communicating pairs — the communication pattern can be arbitrary and input-dependent. Once the communication phase is over, a new round begins. For example, for readers familiar with the MapReduce framework, the computation corresponds to the *reduce* phase, the designation of addressees to the *map* phase, and the actual communication to the *shuffle* phase.

Most previous work has focused on designing efficient algorithms for these systems. To that end, a number of models for MapReduce and its variants have been proposed; see Section 1.2 for a complete list and a detailed comparison to the present work. Since these works aspire to realizable positive results, their models aim to be faithful to a specific system, and often involve a number of system-dependent parameters and constraints that govern the algorithmic design space.

In this work, motivated by the rapidly changing landscape of the systems deployed in practice and in search of impossibility results, we take a different approach. Instead of looking for a faithful model for the system du jour, we extract the most fundamental constraints shared by all of these models and examine the consequent restrictions on feasible computations. By focusing on a single parameter, namely the *input size* for each machine, and a single benchmark, the number of *rounds* of computation, we obtain parameterized lower bounds that apply to every computing framework, present or future, that shares the same design principles.

1.1 Summary of Contributions

Our first contribution is the definition of the s–SHUFFLE model of massively parallel computation, which captures the core properties common to all modern parallel processing systems. Why do we need yet another model of parallel computation? There is a long tradition in theoretical computer science of using one (reasonably realistic) computational model to design algorithms for a problem and a second (stronger and less realistic) model to prove lower bounds. For example, in the field of streaming algorithms, ideal upper bounds are for algorithms that make one pass over the input and use small space and minimal processing time, while lower bounds are usually derived in the (stronger) communication complexity model [2]. In the study of data structures, positive results take care to quantify preprocessing time and query time in addition to space, while space-time trade-offs are typically derived from lower bounds in the extremely powerful cell-probe model [44]. The point of the s–SHUFFLE model is to act as an ultra-powerful model of parallel computation, akin to the communication complexity and cell-probe models, in which lower bounds are unassailable.

Conceptually, computation in the s–SHUFFLE model proceeds in synchronous rounds. In each round, each machine performs an arbitrary computation on its input and sends arbitrary information to arbitrary machines in the next round, subject only to the constraint that each machine receives at most s bits each round. In the most powerful *unbounded-width* version of our model, where the number of machines is unlimited, the only restriction is that the "fan-in" of each machine is limited to s bits, where s is a parameter smaller than the input size n (unlimited extra space can be used for computations on these inputs).

Second, we prove that computations in the s–SHUFFLE model that use few rounds can be represented as low-degree polynomials over the reals. Specifically, we prove that if a function can be computed by such a computation with fan-in s per machine in r rounds (even with unbounded width), then the function has a polynomial representation with degree at most s^r (Theorem 3.1). In particular, computing a function with a polynomial representation of degree n requires $\lceil \log_s n \rceil$ rounds. This lower bound implies, for example, that computing such a function with $s = n^{\epsilon}$ per machine requires $1/\epsilon$ rounds in our model. Similarly, if s is only polylogarithmic, then $\Omega(\log n / \log\log n)$ rounds are needed. Our lower bounds also extend, with a constant-factor degradation, to randomized computations.

Are there super-constant lower bounds on the number of rounds required to compute natural functions when the fan-in s is polynomial in n, such as $s = \sqrt{n}$? Our third contribution is a proof that, in two senses, our lower bounds are the best one could hope for. For the unbounded-width model, our lower bound is completely tight, as every function can be computed in at most $\lceil \log_s n \rceil$ rounds. But what if only a polynomial number of machines are allowed? Here, we show that better lower bounds require proving very strong circuit lower bounds. Specifically, any lower bound asymptotically larger than $\Theta(\log_s n)$ for a function in P would separate NC^1 from P, a notorious open question in circuit complexity (Theorem 6.1).

Fourth, by relating MapReduce-type computations to polynomials, we can apply the sophisticated toolbox known for polynomials to reason about these computations. As one example, by combining the Rivest-Vuillemin solution to the Aanderaa-Rosenberg conjecture [37] with known relationships between decision tree complexity and polynomial degree, we obtain a lower bound of approximately $\frac{1}{2}\log_s n$ on the number of rounds required for deciding any non-trivial monotone graph property, where n is number of vertices and the input is given as the characteristic vector of the graph's edge set (Theorem 4.3).

Finally, we develop new machinery for proving lower bounds on the polynomial degree of Boolean functions, and hence on the round complexity of massively parallel computations. While tight polynomial degree bounds are of course known for many Boolean functions, we prove new (tight) lower bounds for graph problems, including undirected ST-CONNECTIVITY (Theorem 5.5). These degree lower bounds imply lower bounds of roughly $2\log_s n$ on the round complexity of connectivity problems in any conceivable MapReduce-type system, even when the width is unbounded (Corollaries 5.4, 5.6, 5.8, and 5.10).

1.2 Related Work

The MRC model and its precursors. MapReduce was introduced as a system for large scale data processing by Dean and Ghemawat [10]. Karloff et al. [24], inspired in part by Feldman et al. [15], introduced a computational model specially for MapReduce-type computations. This model is called *MRC* and it identified the number of synchronous rounds as the key metric for comparing different algorithms. Furthermore, they limited the amount of parallelism allowed in the model, insisting that it be not too small, by restricting the memory of each machine to be sublinear in the input, but also not too large, by restricting the total number of machines to be sublinear in the input as well. They showed how to simulate a subclass of EREW algorithms in MRC, but left open the question of whether all NC languages can be simulated. Goodrich et al. [19] further extended these simulation results and gave MRC algorithms for sorting and searching.

The MRC model in [24] only placed upper bounds on the number of machines and the space available on each, limiting both to $n^{1-\epsilon}$ for an input of size n and some fixed $\epsilon > 0$. Goodrich et al. [19] were the first to focus on the total input to each machine, the key parameter of the model of the present work. This measure was subsequently adopted by several authors in proving upper bounds on the space-round trade-offs in these computations, and this parameter has been variously called memory, space, and key-complexity [3, 6, 17, 29, 35]. The algorithmic tools and techniques developed for efficient computation in this model include notions of filtering [29], multi-round sampling [14, 28], and coresets [4, 31], among others. Recently Fish et al. [16] introduced a uniform version of MRC, and proved strict hierarchy theorems with respect to the computation time per processor.

Valiant's bulk-synchronous parallel (BSP) model [41] anticipated many of the features of MapReduce-type computations. Computation in the BSP model proceeds in "supersteps," analogous to rounds in the MRC model, with each processor sending and receiving at most h messages between rounds (so h corresponds to our fan-in parameter s). Positive results in the BSP model generally focus on minimizing a combination of delays from synchronization and the amount of computation time performed. One interesting exception is the work of Goodrich [18], who gave asymptotically matching upper and lower bounds on the number of supersteps for sorting in the BSP model. The lower bound in [18] uses a reduction to sorting from computing the logical OR function, and adapts an argument from Cook et al. [9] for "ideal PRAMs" to prove a lower bound of $\Omega(\log_h n)$, where n is the input length. This lower bound is analogous to our warm-up result in Corollary 4.1, where we prove a tight lower bound of $\lceil \log_s n \rceil$ on the number of rounds need to compute the OR function (among others) in our model.

Lower bounds for MapReduce-type computations. Most of the known lower bounds for explicit functions in models of massively parallel computation concern either communication (with a fixed number of rounds) or restricted classes of algorithms (for round lower bounds). Pietracaprina et al. [35] prove non-trivial lower bounds for matrix multiplication in a limited setting, which requires computing all elementary products (and specifically excludes Strassen-like algorithms). Similar kinds of limitations are required by [22] and [6] to prove lower bounds for a list ranking problem and relational query processing, respectively. Bilardi et al. [7] prove tight lower bounds on the fan-in required to compute the Fast Fourier Transform in a restricted BSP model, where the computation conforms to a directed acyclic graph on the operations to be performed. Finally, both Beame et al. [6] and Afrati et al. [1] study, for a fixed number of rounds (usually a single round), space-communication trade-offs.

In distributed computing, the total amount of communication is often the most relevant complexity measure. For example, Woodruff and Zhang [43] and Klauck et al. [25] identify models and problems for which there is no algorithm that beats the communication benchmark of sending the entire input to a single machine. Because massively parallel systems are designed to send a potentially large amount of data in a single round, such communication lower bounds do not generally imply lower bounds for round complexity. For example, in practice, matrix multiplication is regarded as an "easy" problem in MapReduce — indeed, MapReduce was invented for precisely such computations (see e.g. [30]) — even though its solution requires total communication proportional to the size of the matrix.

We next compare and contrast our s–SHUFFLE model with several other parallel and distributed computational models that have been studied.

Congested clique. The "congested clique" model is the special case of the CONGEST model [34] in which each pair of machines is connected by a direct link (see [13] and the references therein). The original motivation for this model was the design and analysis of distributed algorithms, but it is also relevant to massively parallel computation. In the most commonly studied version of the model, n machines with unlimited computational power communicate in synchronized rounds. Each machine initially holds some number m of input bits, and in each round each machine can send a (different) message of w bits to every other machine. (Typically, $w = \Theta(\log n)$.) Thus, the number nw of bits that a machine can send and receive in each round automatically scales linearly with the number of machines.

In our model, we allow the number s of bits that a machine can receive in a given round to be sublinear in the total number of machines. In this sense, our model is weaker than the congested clique model (when $s = o(n)$).[1] On the other hand, MapReduce-type systems differ from distributed settings in that there is little motivation for restricting the amount of point-to-point communication in a round. In our model, a machine can receive all s of its bits in a round from a single other machine. In this sense, our model is stronger than

[1]For a simple example, suppose the machines want to compute the logical OR of their nm input bits. In the congested clique model, this problem can be solved in 1 round. In our model, when the parameter s is less than n, more than 1 round is required.

the congested clique model (when $s = \omega(w)$). Hegeman and Pemmaraju [20] show that congested clique computations can be simulated by MapReduce computations (in the model in [24]), albeit with space-per-machine proportional to the communication-per-machine in the congested clique computation. As already noted, the latter generally scales with the number of machines, while we are generally interested in space-per-machine smaller than this.

Drucker et al. [13] forge an interesting connection between congested clique computations and circuits. In analogy with our "barrier" result in Section 6, they prove a simulation result implying that even slightly super-constant lower bounds on the round complexity of congested clique computations for a problem in NP would imply better circuit size-depth trade-offs for such problems than are currently known (for unbounded fan-in circuits with unweighted threshold gates or with mod-m gates, e.g. $m = 6$).[2]

Circuits with medium fan-in. Another related model is "circuits with medium fan-in" — arbitrary gates with fan-in that scales with, but is smaller than, the input size — introduced recently in [21]. Our model is stronger than the one in [21], with the most significant difference being that the communication pattern in our model can be input-dependent. Our round complexity lower bounds imply depth lower bounds for the model in [21]. The focus of [21] is on circuit size (rather than depth), so the lower bounds and barriers to lower bounds identified in [21] appear incomparable to ours.

Ideal PRAMs. The much older model of "ideal CREW PRAMs" is also highly relevant to our unbounded-width model. In this model, which was introduced in [9], there is an unbounded number of processors with unbounded computational power and a shared memory. Computation proceeds in synchronous rounds. Each round, each processor can read a single memory cell and write to a single memory cell, subject to the constraint that at each time step at most one processor can write to any given memory cell. Perhaps the most intriguing result in [9] is a parallel algorithm that computes the logical OR function (and, through reductions, many other functions) in strictly fewer than $\log_2 n$ rounds. The key idea in this result is to implicitly transmit extra information by *not* writing to a memory cell. This potential "power of staying silent" is exactly what makes our lower bounds in Section 3 interesting and non-trivial. Cook et al. [9] also prove a lower bound of $\Omega(\log n)$ on the number of rounds required to compute the OR function (and many others). This lower bound translates (via a simulation argument) to a lower bound of the form $\Omega(s^{-1} \log n)$ in our model (in [9] each processor reads only one cell in each round, while in our model each machine receives s bits per round). This implied lower bound is non-trivial only when $s = o(\log n)$, and our lower bound of $\Omega(\log_s n)$ is asymptotically superior for all super-constant s.

The work of [9] also inspired Nisan [32] to introduce the fundamental concept of the "block sensitivity" of a Boolean function (the logarithm of which characterizes the ideal PRAM round complexity, up to a constant factor), which in turn is polynomially related to the decision tree complexity, degree, and approximate degree of the function [32, 33]. Our results in Section 4 rely on this circle of ideas.

Finally, our technique of characterizing parallel computations as polynomials resembles the work of Dietzfelbinger et al. [11], who used related ideas to sharpen the lower bound in [9] by a constant factor for many Boolean functions.[3]

[2]To match our barrier in Section 6 and resolve NC^1 vs. P using the techniques in [13], it seems necessary to prove logarithmic (rather than just super-constant) round lower bounds for congested clique computations.

[3]More broadly, variations of "the polynomial method" have been used previously to prove lower bounds in many fields of complexity theory, including circuit complexity [36, 40] and quantum query complexity [5].

2 The s-Shuffle Model

Section 2.1 develops intuition for our computational model by presenting an example, and highlights some ways in which MapReduce-type computations differ from traditional circuit computations. Section 2.2 formally defines the s–SHUFFLE model. Section 2.3 presents the slightly more general s–SHUFFLE(Σ) model and proves that it is powerful enough to simulate MapReduce computations. Only Section 2.2 is essential for understanding most of the rest of the paper.

2.1 A Warm-Up Example

Before formally defining our model, we study a specific example, adapted from Nisan and Szegedy [33]. The first purpose of this example is to introduce the reader to the abstract s–SHUFFLE model in a concrete way. The second is to build an appreciation for the lower bounds in Sections 3–5 by illustrating the cleverness possible in a parallel computation.[4]

Example 2.1 (Silence Is Golden) Consider the Boolean function $E_{12} : \{0,1\}^3 \to \{0,1\}$ on three inputs which evaluates to 1 if and only if exactly one or two inputs are 1. Define $E_{12}^2 : \{0,1\}^9 \to \{0,1\}$ as the Boolean function that takes nine inputs, applies E_{12} to each block of three inputs, and then applies E_{12} to the results of the three blocks. For instance:

- $E_{12}^2(0,0,1,0,1,0,1,0,0) = E_{12}(1,1,1) = 0$;
- $E_{12}^2(0,0,0,1,1,0,1,1,1) = E_{12}(0,1,0) = 1$.

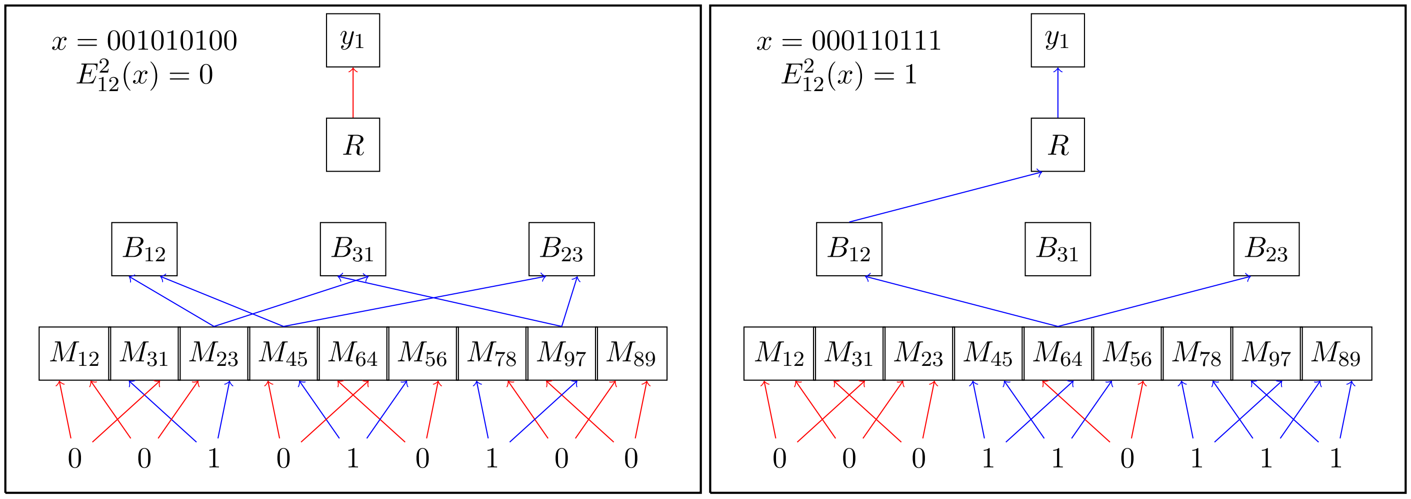

Figure 1: The parallel computation of E_{12}^2, for two different inputs. The arrows show how the machines send bits; an arrow pointing to the bottom left of a machine indicates that the bit is sent to the first port, while the bottom right indicates the second port. Notice that at most one arrow points to a given port, but the arrow that does so may change from input to input. Red arrows represent a zero bit and blue arrows represent a one bit.

[4]Of course, there are countless examples of analogously clever parallel algorithms in the literature.

We now describe a remarkably efficient strategy for computing E_{12}^2 in parallel, using only machines that operate on two (ordered) bits at a time. We first show how to compute E_{12} in two "rounds." Let (x_1, x_2, x_3) denote the input. The first machine reads the bits x_1 and x_2; the second machine x_2 and x_3; and the last machine x_3 and x_1. There is also a fourth machine which belongs to "the second round." Each machine in the first round sends a 1 to the second-round machine if its inputs are 0 and 1 (in this order); otherwise, it stays silent (i.e., sends nothing). The second-round machine receives either a 1 bit (if $E_{12}(x_1, x_2, x_3) = 1$) or no bits at all (if $E_{12}(x_1, x_2, x_3) = 0$), and in all cases can correctly determine and output the value of the function.

To compute the function E_{12}^2, we use a layered version of the same idea (see also Figure 1). We think of the nine input bits as three blocks of three bits each. There are nine machines in the first round, three for each block. There are three machines in the second round, each responsible for a pair of blocks. There is a single machine in the third round, responsible for the final output.

The three bits of a block are distributed to the corresponding three first-round machines (in ordered pairs) as in the computation of E_{12}. Second-round machines have two "ports" in that their inputs are also ordered; each has a "first input" (possibly empty) and a "second input" (again, possibly empty). If the inputs of a first-round machine are 0 and 1 (in this order), then the machine sends a 1 to the two second-round machines responsible for its block; otherwise, it sends nothing. The communication pattern between the three blocks of first-round machines and the second-round machines mirrors that of the distribution of bits to first-round machines: the first second-round machine receives bits (if any) from the first and second blocks of first-round machines on its first and second ports, respectively; the middle second-round machine receives bits from the second and third blocks on its first and second ports, respectively; and the last second-round machine receives bits from the third and first blocks on its first and second ports, respectively. If a second-round machine receives nothing as its first input and a 1 as its second input, then it sends a 1 to the third-round machine; otherwise, it sends nothing. Finally, the third-round machine outputs 1 if it was sent a 1, and otherwise (in which case it receives nothing) it outputs 0. It is straightforward to verify that this computation correctly evaluates E_{12}^2 on all inputs. The computation uses three rounds of communication, with each machine receiving at most two bits of input.

Shuffles vs. circuits. There are, unsurprisingly, many resemblances between this parallel computation and circuits — each picture in Figure 1 looks like a Boolean circuit, with machines corresponding to gates, the number of rounds corresponding to the depth, the number of machines corresponding to the size, and the number of input ports corresponding to the fan-in. We note, however, that no circuit with fan-in 2 and depth 3 (of any size) computes the function E_{12}^2, as the function depends on all nine of its inputs. This is true even for circuits with an alphabet larger than $\{0, 1\}$, such as $\{0, 1, \text{"nothing"}\}$. In general, while $\lceil \log_s n \rceil$ is an obvious lower bound on the depth of any circuit with fan-in s that computes a function that depends on all inputs, this lower bound does not necessarily hold for the number of rounds required by a MapReduce-type computation.

2.2 The Basic Model

What augmentations to standard circuit models are required to capture arbitrary MapReduce-type computations? The first two are evident from our parallel computation of E_{12}^2 in Section 2.1.

(1) The communication pattern, which plays the role of the circuit topology (i.e., which gates are connected to which), can be input-dependent. (Cf., Figure 1.)

(2) Each machine has the option of staying silent and sending nothing. We refer to this as "sending a $\perp$."

At first blush, extension (2) might seem equivalent to enlarging the alphabet by one character. This is not quite correct, since $\perp$'s combine with each other and with other bits in a particular way.

Definition 2.2 ($\perp$-sum) The $\perp$-*sum* of $z_1, z_2, \ldots, z_m \in \{0, 1, \perp\}$ is:

- 1 if exactly one z_i is 1 and the rest are $\perp$;
- 0 if exactly one z_i is 0 and the rest are $\perp$;
- $\perp$ if every z_i is $\perp$;
- undefined (or invalid) otherwise.

The $\perp$-sum of m s-tuples $a_1, \ldots, a_m$ is the entry-by-entry $\perp$-sum, denoted $\odot_{i=1}^{m} a_i$.

Most circuit models severely restrict the computational power of each gate. This is not appropriate in the present context, where each "gate" corresponds to a general-purpose machine embedded in a MapReduce-type infrastructure. This motivates our third extension.

(3) Each machine can perform an arbitrary computation on its inputs.[5]

Our formal model extends the usual notion of a circuit (with fan-in s) to accommodate (1)–(3). Our notation follows that in Vollmer [42] for circuits.

Definition 2.3 (s–SHUFFLE Computation) An *R-round s–*SHUFFLE *computation with inputs $x_1, \ldots, x_n$ and outputs $y_1, \ldots, y_k$* has the following ingredients:

1. A set V of *machines*, which includes one machine for each input bit x_i and each output bit y_i.
2. An assignment of a *round* $r(v)$ to each machine $v \in V$. Machines corresponding to input bits have round 0. Machines corresponding to output bits have round $R + 1$. All other machines have a round in $\{1, 2, \ldots, R\}$.
3. For each pair (u, v) of machines with $r(u) < r(v)$, a function α_{uv} from $\{0, 1, \perp\}^s$ to $\{0, 1, \perp\}^s$.

We think of each machine as having s "ports," where each port can accept at most 1 bit. The interpretation of a function α_{uv} is: given that machine u received $\mathbf{z} = z_1, \ldots, z_s$ on its s input ports (where each $z_i \in \{0, 1, \perp\}$), it sends the message $\alpha_{uv}(\mathbf{z}) \in \{0, 1, \perp\}^s$ to the s input ports of v.[6] Thus, machine u explicitly communicates with machine v (on at least one port) if and only if at least one coordinate of $\alpha_{uv}(\mathbf{z})$ is not $\perp$. We conclude that the model supports input-dependent communication patterns, as in (1). The model also supports (2) and (3), by definition.

The output of an s–SHUFFLE computation is evaluated as in a circuit, with the important constraint that, on every input, each input port should receive a bit (i.e., a non-$\perp$) from at most one machine.

[5]When the goal is to prove algorithmically meaningful upper bounds, it would be sensible to restrict machines to efficient computations. Also, the total space used by a machine, both for its input and for its computations, should be small (certainly sublinear in the total input size). Given our focus on lower bounds, our allowance of arbitrary computations and unlimited scratch space on each machine only makes our results stronger.

[6]The model allows a machine to communicate with machines in all later rounds, not just machines in the next round. Computations of the former type can be translated to computations of the latter type by adding dummy machines, so this is not an important distinction.

Definition 2.4 (Result of an s–SHUFFLE Computation) The *result* of an s–SHUFFLE computation assigns a value $g(v) \in \{0, 1, \perp\}^s$ to every machine $v \in V$, and is defined inductively as follows.

1. For a round-0 machine v, corresponding to an input bit x_i, the value $g(v)$ is the s-tuple $(x_i, \perp, \perp, \ldots, \perp)$.
2. Given the value $g(u)$ assigned to every machine u with $r(u) < q$, the value assigned to a machine v with $r(v) = q$ is the $\perp$-sum, over all machines u with $r(u) < r(v)$, of the message $\alpha_{uv}(g(u))$ sent to v by u:
$$g(v) := \odot_{u\,:\,r(u)<r(v)} \alpha_{uv}(g(u)). \tag{1}$$

The result of an s–SHUFFLE computation is *valid* if every $\perp$-sum in equation (1) is well defined, and, if for every machine v corresponding to an output bit y_i, the value $g(v)$ is either $(0, \perp, \perp, \ldots, \perp)$ or $(1, \perp, \perp, \ldots, \perp)$. The "0" or "1" in the first coordinate is then interpreted as the corresponding output bit y_i. Unless otherwise noted, we consider only valid s–SHUFFLE computations.

By convention, the machines corresponding to the input and output bits of the function do not contribute to the number of rounds — these are "placeholder machines" that cannot do any non-trivial computations. Translating the parallel computations in Section 2.1 of the functions E_{12} and E_{12}^2 into the formalism of Definitions 2.3 and 2.4 yields 2-round and 3-round s–SHUFFLE computations, respectively, in accordance with intuition.

The following definition is analogous to that for circuits.

Definition 2.5 (Width) The *width* of an s–SHUFFLE computation is the maximum number of machines in a round other than round 0 and the final round.

Remark 2.6 (Randomized Computations) A randomized s–SHUFFLE computation is a probability distribution over deterministic s–SHUFFLE computations.[7] We say that such a computation computes a function f if, for every input $\mathbf{x}$, the output of the computation equals $f(\mathbf{x})$ with probability at least $2/3$.

2.3 Simulating MapReduce in the s–SHUFFLE Model

In the definition of the s–SHUFFLE model, the input ports of each machine are ordered, and a message from one machine to another includes a designation of which outbound bit is destined for which input port. In actual MapReduce computations, in each round a machine receives an unordered collection of key-value pairs sent from other machines. We next extend the basic s–SHUFFLE model to accommodate such unordered sets of messages.

Fix a word size w, the largest message size (in bits) that one machine can send to another. (In our MapReduce simulation, w will be $n^{1-\epsilon}$ for a parameter $\epsilon > 0$.) Let Σ denote all possible multi-sets of bit strings with length at most w. The s–SHUFFLE(Σ) model is defined analogously to the s–SHUFFLE model, with the following changes. The value $g(u)$ assigned to a machine u, other than the machines corresponding to the inputs and the outputs, is now an element of Σ with total bit complexity (i.e., sum of message lengths) at most s. Thus, a machine now receives an unordered set of messages with at most w bits each, but is still constrained to receive at most s bits of information per round. We redefine the functions α_{uv} to have domain and range equal to the elements of Σ that have total bit complexity at most s. A $\perp$-sum of such multi-sets is

[7]This definition corresponds to "public coins," in that it allows different machines to coordinate their coin flips with no communication. One could also define a "private coins" model where each machine flips its own coins. Our lower bounds are for the stronger public-coin model, and apply to the private-coin model as a special case.

their union (with multiplicities adding), and is undefined if the total bit complexity of this union exceeds s. The value of the ith round-0 machine is $\{x_i\}$, where x_i is the ith input bit. In a valid s–SHUFFLE(Σ) computation, the value of every machine corresponding to an output bit is either $\{0\}$ or $\{1\}$.

With this extension to the model, it is straightforward to map an arbitrary r-round MapReduce computation that uses m machines with space at most s bits each to an $(r+1)$-round s–SHUFFLE(Σ) computation that uses $m(r+1)$ machines (not counting the machines that correspond to input and output bits). To be more precise, we focus specifically on the computational model used to define the $\mathcal{MRC}$ complexity class in [24]. In this model, there is a parameter $\epsilon > 0$, and there are at most $n^{1-\epsilon}$ machines, each with space at most $n^{1-\epsilon}$ bits. The basic unit of information in a MapReduce computation is a $\langle key; value\rangle$ pair. The computation in a given round is defined by two sets of functions, mappers and reducers. A mapper is a function that takes as input some number of $\langle key; value\rangle$ pairs and, independently for each such pair, produces a finite multiset of $\langle key; value\rangle$ pairs. A reducer is a function that takes as input a binary string, representing a key k, and a set of values $v_1, v_2, \ldots$ (from key-value pairs with key k), and outputs a set of $\langle key; value\rangle$ pairs. We can think of the set of reducers as being indexed by the corresponding key k. The shuffle step of a round is responsible for routing the key-value pairs generated by the mappers in that round to the appropriate reducers in that round, essentially by grouping and assigning together key-value pairs that have the same key. The key-value pairs emitted by reducers proceed straight to the mappers of the next round; no routing between machines is necessary. It should be possible to deduce the output of the computation from the key-value pairs produced by the reducers in the final round. Every machine used in the MapReduce computation can host any number of mappers and reducers, subject to the per-machine space constraint of $n^{1-\epsilon}$ bits.

Proposition 2.7 (Simulating MapReduce) Every r-round MapReduce computation with m machines and space s per machine can be simulated by an $(r+1)$-round s–SHUFFLE(Σ) computation with $(r+1)m$ machines and word size s.

Proof: The idea is to make the machines in a round of the s–SHUFFLE(Σ) computation responsible for simulating the computation and communication that occurs in the corresponding round of the MapReduce computation.

In detail, fix a MapReduce computation that uses m machines with space at most s each. Call these machines *M-machines*, to distinguish them from the machines used in our s–SHUFFLE(Σ) computation, which we call *S-machines*. We simulate the MapReduce computation in the s–SHUFFLE(Σ) model as follows. In addition to the usual (fictitious) S-machines corresponding to input and output bits, there are m S-machines for each round $1, 2, \ldots, r+1$. The functions α_{uv} from round-0 S-machines u to round-1 S-machines v are defined to simulate the initial assignment of input bits to the m M-machines of the MapReduce computation. In a generic round of the s–SHUFFLE(Σ) simulation, every round-i S-machine simulates the round-$(i-1)$ reducers and the round-i mappers of the corresponding M-machine, and also the routing of key-value pairs to reducers done in the ith shuffle step of the MapReduce computation. (For $i = 1$, round-1 S-machines only need to simulate the corresponding round-1 mappers and the subsequent shuffle step.) Thus for round-i and round-$(i+1)$ S-machines u and v, the function α_{uv} is defined as the set of key-value pairs generated by the M-machine corresponding to u (by its round-$(i-1)$ reducers composed with its round-i mappers, given the key-value pairs it was assigned in the previous round) that get routed to the M-machine corresponding to v in the shuffle step of round i of the MapReduce computation. Because the M-machines have space only s, the total bit complexity of the messages sent to an S-machine in the simulation is at most s.

Inductively, the value $g(u)$ assigned to an S-machine u in round i of the s–SHUFFLE(Σ) computation is exactly the set of key-value pairs given to the round-$(i-1)$ reducers hosted on the corresponding M-machine in the MapReduce computation. Thus the round-$(r+1)$ S-machines are assigned the final key-value pairs of the MapReduce computation, from which the output bits can be deduced and sent to the corresponding output S-machines. This completes the simulation of MapReduce computations in the s–SHUFFLE(Σ) model. ■

3 Representing Shuffles as Polynomials

This section shows that low-round s–SHUFFLE computations with small s can only compute Boolean functions that can be represented as low-degree polynomials. Section 3.1 proves this for the basic s–SHUFFLE model. Section 3.2 shows that the implied lower bounds on round complexity are the best possible for the unbounded-width s–SHUFFLE model. Section 3.3 extends these lower bounds to randomized computations and Section 3.4 to the s–SHUFFLE(Σ) model from Section 2.3.

3.1 Representing s–SHUFFLE Computations

To prove lower bounds on the number of rounds required by an s–SHUFFLE computation, we associate each computation with a polynomial over the reals that matches (on $\{0,1\}^n$) the function it computes. We show that the number of rounds used by the computation governs the maximum degree of this polynomial, and hence functions that correspond to polynomials of high degree cannot be computed in few rounds.

Theorem 3.1 (Round-Efficient Shuffles Are Low-Degree Polynomials) *Suppose that an s–SHUFFLE computation correctly computes the function $f : \{0,1\}^n \to \{0,1\}^k$ in r rounds. Then, there are k polynomials $\{p_i(x_1,\ldots,x_n)\}_{i=1}^k$ of degree at most s^r such that $p_i(\mathbf{x}) = f(\mathbf{x})_i$ for all $i \in \{1,2,\ldots,k\}$ and $\mathbf{x} \in \{0,1\}^n$.*

Proof: We proceed by induction on the number of rounds. We claim that for every non-output machine $v \in V$ and value $\mathbf{z} \in \{0,1,\perp\}^s$, there is a polynomial $p_{v,\mathbf{z}}(x_1,\ldots,x_n)$ that evaluates to 1 on points $\mathbf{x}$ for which the computation's assigned value $g(v)$ to v is $\mathbf{z}$ and to 0 on all other points $\mathbf{x} \in \{0,1\}^n$. Furthermore, $p_{v,\mathbf{z}}$ has degree at most $s^{r(v)}$.

The base case of the induction is for a machine v in round zero ($r(v) = 0$). Each such machine corresponds to an input bit x_i and its value $g(v)$ is $(x_i,\perp,\perp,\ldots,\perp)$. The (degree-1) polynomials for $\mathbf{z} = (0,\perp,\ldots,\perp)$ and $\mathbf{z} = (1,\perp,\ldots,\perp)$ are $p_{v,\mathbf{z}}(x_1,\ldots,x_n) = 1-x_i$ and $p_{v,\mathbf{z}}(x_1,\ldots,x_n) = x_i$, respectively. All other values of $\mathbf{z}$ are impossible for such a machine, so they have polynomials $p_{v,\mathbf{z}}(x_1,\ldots,x_n) = 0$.

Consider a machine v that corresponds to neither an input bit nor an output bit. Fix some potential value $\mathbf{z} \in \{0,1,\perp\}^s$ of $g(v)$, and focus first on the i^{th} coordinate of $g(v)$. When is it equal to z_i? We first consider the case where z_i is 0 or 1 (and not $\perp$). The entry $g(v)_i$ is a $\perp$-sum of the messages that machine v receives on port i from all machines in previous rounds. Hence $g(v)_i = z_i$ when a single machine in a previous round sends z_i on port i to machine v (and the rest send $\perp$'s). Consider some machine u of a previous round. It sends z_i on port i to machine v when $\alpha_{uv}(g(u))$ has an i^{th} entry of z_i. The set of assigned values $g(u)$ to u that cause it to do this is the preimage, under α_{uv}, of the subset of $\{0,1,\perp\}^s$ containing all elements with a z_i in the ith coordinate. By our inductive hypothesis, for each value $g(u)$ in this preimage, there is a polynomial of degree at most $s^{r(v)-1}$ that indicates the inputs $\mathbf{x}$ for which u receives this value. Since u is assigned exactly one value, we can sum these polynomials together to get a polynomial (of degree at most $s^{r(v)-1}$) that indicates when u receives any value in this preimage. This polynomial indicates the inputs $\mathbf{x}$ for which u sends z_i to port i of machine v. Furthermore, since the s–SHUFFLE computation is valid, at most one machine in a previous round can send a non-$\perp$ value to v on port i. Summing over the polynomials of all machines in previous rounds yields a polynomial, still with degree at most $s^{r(v)-1}$, that indicates whether or not v received z_i on port i (from any machine).

The inputs $\mathbf{x}$ for which $g(v)_i = \perp$ are those for which no machine from a previous round sends a 0 or a 1 to v on the ith port. A polynomial for this case is just 1 minus the polynomials for the $g(v)_i = 0$ and $g(v)_i = 1$ cases. The degree of this polynomial is at most $s^{r(v)-1}$.

To obtain a polynomial that represents the inputs $\mathbf{x}$ for which $g(v) = \mathbf{z}$, we just take the product of the polynomials that represent the events $g(v)_1 = z_1, \ldots, g(v)_s = z_s$. This polynomial has degree at most $s^{r(v)}$, and this completes the inductive step.

Finally, consider a machine v that corresponds to an output bit y_i. As in the inductive step above, we can represent the first coordinate of $g(y_i)$ with a polynomial p_i of degree at most $s^{r(y_i)-1} = s^r$. Since the s–SHUFFLE computation is valid, the polynomial p_i also represents the ith output bit of the function f. ■

The following corollary is immediate.

Corollary 3.2 (Round Lower Bound) *If some output bit of the function $f : \{0,1\}^n \to \{0,1\}^k$ cannot be represented by a polynomial with degree less than d, then every s–SHUFFLE computation that computes f uses at least $\lceil \log_s d \rceil$ rounds.*

There is a mature toolbox for bounding below the polynomial degree necessary to represent Boolean functions; we apply and contribute to this toolbox in Section 4.

Theorem 3.1 and Corollary 3.2 apply even to unbounded-width s–SHUFFLE computations. For functions f with $d = n$, Proposition 3.4 gives a matching upper bound in the unbounded-width model.

Remark 3.3 (No Converse to Theorem 3.1) A natural question (raised by a referee) is whether the converse to Theorem 3.1 also holds: can *every* degree-s^r polynomial be computed by an r-round s–SHUFFLE? The answer is no. One simple counterexample is the degree-3 polynomial

$$p(x_1,x_2,x_3,x_4) = (x_1+x_2+x_3+x_4) - (x_1x_2+x_1x_3+x_1x_4) - 2(x_2x_3+x_2x_4+x_3x_4) \\ + (x_1x_2x_3+x_1x_2x_4+x_1x_3x_4) + 3x_2x_3x_4$$

in four variables. Restricted to the set $\{0,1\}^4$, p is a Boolean function, equal to 1 on every point (x_1,x_2,x_3,x_4) with Hamming weight 1 or 4, and also every point with Hamming weight 2 for which $x_1 = 1$. A simple case analysis shows that there is no 1-round 3-shuffle computation that computes this function. We leave as an open question the problem of characterizing the set of functions that can be computed by r-round s–SHUFFLE computations.

3.2 A Matching Upper Bound in Unbounded Width Model

We next show that the round lower bound in Corollary 3.2 is best possible in the unbounded-width s–SHUFFLE model (with an unlimited number of machines): for every $s \geq 2$ and every function $f : \{0,1\}^n \to \{0,1\}^k$, f can be computed by an s–SHUFFLE computation in $\lceil \log_s n \rceil$ rounds. We conclude that stronger lower bounds are only possible under additional assumptions. The prospects for stronger lower bounds for polynomial-width s–SHUFFLE computations are the subject of Section 6.

Proposition 3.4 (Unbounded Width Upper Bound) For every $s \geq 2$, every function $f : \{0,1\}^n \to \{0,1\}^k$ can be computed by an s–SHUFFLE computation in $\lceil \log_s n \rceil$ rounds.

Proof: The idea is for every possible input to have a dedicated set of machines, responsible for computing f on that input. In more detail, there is one machine $v_{\mathbf{x}}$ at round $\lceil \log_s n \rceil$ – the last round before the outputs –

for each possible input $\mathbf{x} \in \{0,1\}^n$. Each machine $v_{\mathbf{x}}$ serves as the root of a tree $T_{\mathbf{x}}$ with branching factor s and depth $\lceil \log_s n \rceil$. The machines of $T_{\mathbf{x}}$ are responsible for notifying $v_{\mathbf{x}}$ whether or not the input is in fact $\mathbf{x}$. Each of the $\lceil n/s \rceil$ machines at the leaves checks if the s input bits that it monitors conforms to $\mathbf{x}$, and if so, it notifies its parent accordingly. After $\lceil \log_s n \rceil$ levels of such computations, $v_{\mathbf{x}}$ knows whether or not the input is indeed $\mathbf{x}$. The machine $v_{\mathbf{x}}$ either sends nothing to the outputs (in the likely case that the input is not $\mathbf{x}$), or the bits of the answer $f(\mathbf{x})$ to the k output machines (if the input is $\mathbf{x}$). ■

3.3 Randomized Computations

We say that a polynomial $p(x_1, \ldots, x_n)$ *approximately represents* a Boolean function $f : \{0,1\}^n \to \{0,1\}$ if $|p(\mathbf{x}) - f(\mathbf{x})| \leq \frac{1}{3}$ for every $\mathbf{x} \in \{0,1\}^n$. The *approximate degree* of a Boolean function is the smallest value of d such that f can be approximately represented by a degree-d polynomial. We next give an analog of Theorem 3.1 for randomized s–SHUFFLE computations (Remark 2.6), which connects the rounds required by such computations to the approximate degree of Boolean functions.

Theorem 3.5 (Round-Efficient Randomized Shuffles Are Low-Degree Approximate Polynomials) *Suppose that a randomized s–*SHUFFLE *computation computes the function $f : \{0,1\}^n \to \{0,1\}^k$ in r rounds. Then there are k polynomials $\{p_i(x_1, \ldots, x_n)\}_{i=1}^k$ of degree at most s^r such that $|p_i(\mathbf{x}) - f(\mathbf{x})_i| \leq \frac{1}{3}$ for all $i \in \{1, 2, \ldots, k\}$ and $\mathbf{x} \in \{0,1\}^n$.*

Proof: A randomized s–SHUFFLE computation is a distribution over deterministic computations. By Theorem 3.1, we can represent each output bit of these deterministic computations by a polynomial of degree at most s^r. The weighted averages of these polynomials, with the weights equal to the probabilities of the corresponding deterministic computations, yields polynomials $p_1, \ldots, p_k$ such that, for every i and $\mathbf{x} \in \{0,1\}^n$, $p_i(\mathbf{x})$ equals the probability that the randomized s–SHUFFLE computation outputs a 1 on the input $\mathbf{x}$. Since the randomized computation computes f in the sense of Remark 2.6, these polynomials satisfy the conclusion of the theorem. ■

Corollary 3.6 (Round Lower Bound (Randomized)) *If some output bit of the function $f : \{0,1\}^n \to \{0,1\}^k$ has approximate degree at least d, then every randomized s–*SHUFFLE *computation that computes f uses at least $\lceil \log_s d \rceil$ rounds.*

Section 4 applies tools for bounding from below the approximate degree of Boolean functions to derive round lower bounds for randomized s–SHUFFLE computations.

3.4 Representing the s–SHUFFLE(Σ) Model

Finally, we extend Theorem 3.1 to the s–SHUFFLE(Σ) model of Section 2.3.

Theorem 3.7 (Extension to the s–SHUFFLE(Σ) Model) *Suppose that an s–*SHUFFLE*(Σ) computation correctly computes the function $f : \{0,1\}^n \to \{0,1\}^k$ in r rounds. Then, there are k polynomials $\{p_i(x_1, \ldots, x_n)\}_{i=1}^k$ of degree at most s^r such that $p_i(\mathbf{x}) = f(\mathbf{x})_i$ for all $i \in \{1, 2, \ldots, k\}$ and $\mathbf{x} \in \{0,1\}^n$.*

Proof: The proof follows that of Theorem 3.1, with minor changes. We will again have a polynomial $p_{v,\mathbf{z}}(x_1, \ldots, x_n)$ that indicates the inputs $\mathbf{x}$ for which machine v is assigned the value $\mathbf{z}$. Note that $\mathbf{z}$ is now an element of Σ—a multi-set of strings each with length at most w—with total bit complexity at most s, rather than an element of $\{0,1,\perp\}^s$. We define degree-1 polynomials for the input machines as in the proof of Theorem 3.1.

For the general case, consider a machine v that corresponds to neither an input bit nor an output bit. We can no longer argue coordinate-by-coordinate. First consider an input $\mathbf{z}$ with maximum-possible bit complexity s. For a given assignment of the messages in $\mathbf{z}$ to (at most s) machines in rounds prior to $r(v)$, the event that v receives the value $\mathbf{z}$ in precisely this way can be inductively expressed as the product of at most s polynomials that have degree at most $s^{r(v)-1}$ each, yielding a polynomial with degree at most $s^{r(v)}$. (Because $\mathbf{z}$ has the maximum-allowable size s and the computation is valid, the other machines must send nothing to v.) Summing over all possible assignments of $\mathbf{z}$ to previous machines yields a degree-$s^{r(v)}$ polynomial that indicates the inputs for which v is assigned the value $\mathbf{z}$.

Now consider a value $\mathbf{z}$ with bit complexity only $s-1$. The event that v receives *some superset of* $\mathbf{z}$ can be likewise expressed as a polynomial with degree at most $s^{r(v)}$. Subtracting out the events corresponding to strict supersets of $\mathbf{z}$—which much have bit complexity exactly s and hence are each representable by an $s^{r(v)}$-degree polynomial—gives a degree-$s^{r(v)}$ polynomial that indicates the inputs for which v is assigned the value $\mathbf{z}$. Continuing the argument with values $\mathbf{z}$ with $s-2$ bits, and so on, gives a degree-$s^{r(v)}$ polynomial representation for every v and $\mathbf{z}$.

Finally, in a valid s–SHUFFLE(Σ) computation, an output machine receives either a single message of $\{0\}$ or a single message of $\{1\}$. Indicating which of these is the case can be expressed as a sum of polynomials from the last non-output round of the computation, and hence also has a s^r-degree polynomial representation. ■

Corollary 3.8 (Round Lower Bound (s–SHUFFLE(Σ) Model)) *If some output bit of the function $f : \{0,1\}^n \to \{0,1\}^k$ cannot be represented by a polynomial with degree less than d, then every s–SHUFFLE(Σ) computation that computes f uses at least $\lceil \log_s d \rceil$ rounds.*

By the simulation argument in Proposition 2.7, the same lower bound (less 1) applies to the computational model of MapReduce used to define the $\mathcal{MRC}$ complexity class in [24].

4 Lower Bounds for Polynomial Degree

Corollary 3.2 and its extensions reduce the problem of proving lower bounds on the round complexity of s–SHUFFLE computations to proving lower bounds on the degree of polynomials that exactly or approximately represent the function to be computed. There is a sophisticated set of tools for proving the latter type of lower bounds, which we now put to use.

4.1 Warm Up

Consider a Boolean function $f : \{0,1\}^n \to \{0,1\}$. If f can be represented by a polynomial p, meaning $f(\mathbf{x}) = p(\mathbf{x})$ for all $\mathbf{x} \in \{0,1\}^n$, then it can be represented by a multilinear polynomial (since $x_i^2 = x_i$ for $x_i \in \{0,1\}$). Recall that for every such function f, there is a unique multilinear polynomial that represents it (see e.g. [12]). We call the degree of this polynomial the *degree* of the Boolean function. The maximum-possible degree is n.

For example, since the AND_n function is represented (uniquely) by the polynomial $\prod_{i=1}^n x_i$, it has the maximum-possible degree. Similarly, the OR_n function has degree n because it is represented by the polynomial $1 - \prod_{i=1}^n (1 - x_i)$. Corollary 3.2 immediately implies the following for s–SHUFFLE computations.

Corollary 4.1 (Lower Bound for AND_n and OR_n) *Every s–SHUFFLE computation that computes the AND_n or the OR_n function uses at least $\lceil \log_s n \rceil$ rounds.*

Corollary 3.8 implies the same lower bound for s–SHUFFLE(Σ) computations.

4.2 Monotone Graph Properties

While pinning down the precise degree of a Boolean function representing a graph problem can be a difficult task, there are powerful tools for proving loose but useful lower bounds.

The first ingredient concerns the decision tree complexity of monotone graph properties. Recall that a graph property is a property of undirected graphs that is independent of the vertex labeling. It is non-trivial if it does not assign the same value to all graphs, and monotone if adding edges cannot destroy the property. The Aanderaa-Rosenberg conjecture states that, for every non-trivial monotone graph property, the decision-tree complexity is $\Omega(n^2)$ [38]. It was first proved by Rivest and Vuillemin with a lower bound of $n^2/16$ [37], and a long line of work has yielded a lower bound of $n^2/3 - o(n^2)$ [23, 26, 27, 39].

The second ingredient is the known polynomial relationship between the decision tree complexity and the degree of a Boolean function. Specifically, Nisan and Smolensky (cited in [8]) proved that, for every Boolean function f, the decision-tree complexity of f is at most $2\deg(f)^4$, where $\deg(f)$ is the degree of f.

Combining these two ingredients with Corollary 3.2 yields the following.

Theorem 4.2 (Lower Bound for Monotone Graph Properties) *For every non-trivial monotone graph property $f : \{0,1\}^{\binom{n}{2}} \to \{0,1\}$ of graphs with n vertices, every s–SHUFFLE computation that computes f requires at least $\frac{1}{2}\log_s n - \frac{1}{4}\log_s 6$ rounds.*

In Section 5 we prove tight degree lower bounds for polynomials that represent classical connectivity problems (special cases of monotone graph properties), improving the lower bound in Theorem 4.2 by roughly a factor of 4 (Corollaries 5.4, 5.6, 5.8, and 5.10).

4.3 Approximate Degree and Randomized Computations

Recall from Corollary 3.6 that lower bounds on the approximate degree of a Boolean function translate to lower bounds on the number of rounds required by randomized s–SHUFFLE computations. It is also known that both the degree and decision tree complexity of every Boolean function are polynomially related to its approximate degree: $\deg(f) \le D(f) \le 216 \cdot \widetilde{\deg}(f)^6$, where $\widetilde{\deg}$ denotes the approximate degree [5, 33].[8] We therefore have the following analog of Theorem 4.2 for randomized computations.

Theorem 4.3 (Lower Bound for Monotone Graph Properties (Randomized)) *For every non-trivial monotone graph property $f : \{0,1\}^{\binom{n}{2}} \to \{0,1\}$ of graphs with n vertices, every randomized s–SHUFFLE computation that computes f requires at least $\frac{1}{3}\log_s n - \frac{1}{6}\log_s 648$ rounds.*

In general, round lower bounds for deterministic s–SHUFFLE computations proved via Corollaries 3.2 and 3.8 extend automatically, with a small constant-factor loss, to randomized computations.

5 Lower Bounds for Graph Computations

The goal of this section is develop and apply a technique for showing that many problems that are important in the field of algorithms (as opposed to in Boolean function analysis) — such as graph connectivity problems — have maximum-possible degree.

[8] For some specific functions, better bounds are known. For example, the majority function has approximate degree $\Omega(\sqrt{n})$ [36].

Buhrman and de Wolf [8] credit Yaoyun Shi with the observation that a Boolean function has degree n if and only if the number of even solutions (i.e., assignments with an even number of 1s for which the function evaluates to 1) is not equal to the number of odd solutions. For the reader familiar with Boolean Fourier analysis, this corresponds to computing whether $\hat{f}([n])$, the function's Fourier coefficient for the set $[n]$, is nonzero. For example, the function XOR_n has no even solutions and 2^{n-1} odd solutions, so the function has degree n and hence (by Corollary 3.2) s–SHUFFLE computations require $\lceil \log_s n \rceil$ rounds to compute it.

5.1 Parity Difference Preliminaries

We first establish some notation and lemmas that simplify the proofs.

Definition 5.1 (Parity Difference) Given a set $S \subseteq \{0,1\}^n$, define the *parity difference function* as the number of even inputs in S minus the number of odd inputs in S:

$$\Phi(S) = \sum_{x \in S} \prod_i (-1)^{x_i}.$$

Define the *parity difference* of a Boolean function f on n bits by $\Phi(f) = \Phi(\{x \mid f(x) = 1\})$.

In Fourier-analytic terms, $\Phi(f)$ is exactly $2^n \hat{f}([n])$. We have the following identities.

Lemma 5.2 (Properties of Parity Differences) *Fix a set $S \subseteq \{0,1\}^n$.*

(a) If S_1, S_2 form a partition of S, then

$$\Phi(S) = \Phi(S_1) + \Phi(S_2).$$

(b) If S is the Cartesian product $S_1 \times S_2$ of sets S_1, S_2, then

$$\Phi(S) = \Phi(S_1) \cdot \Phi(S_2).$$

Proof: The lemma follows immediately from basic Boolean Fourier analysis, if we consider the functions that represent the characteristic vectors of S, S_1, and S_2.

It is also easy to prove the lemma directly from definitions. For part (a), using the partition assumption, we have

$$\sum_{x \in S} \prod_i (-1)^{x_i} = \sum_{y \in S_1} \prod_i (-1)^{y_i} + \sum_{z \in S_2} \prod_i (-1)^{z_i}.$$

For part (b), we can split each $x \in S$ according to the product $S_1 \times S_2$ to factor the sum:

$$\begin{aligned} \sum_{x \in S} \prod_i (-1)^{x_i} &= \sum_{y \in S_1} \sum_{z \in S_2} \prod_i (-1)^{y_i} \prod_i (-1)^{z_i} \\ &= \sum_{y \in S_1} \prod_i (-1)^{y_i} \sum_{z \in S_2} \prod_i (-1)^{z_i}. \end{aligned}$$

■

5.2 Graph Connectivity Problems

We express graph problems as Boolean functions using a $\binom{n}{2}$-variable input to represent an undirected graph on n vertices (or a $2\binom{n}{2}$-variable input for directed graphs), where each variable indicates the presence or absence of a given edge. This is effectively the adjacency matrix representation of the graph.

The CONNECTIVITY problem is: given a graph $G = (V, E)$, is there a path from every vertex to every other vertex? The related ST-CONNECTIVITY problem is: given a graph $G = (V, E)$ and two vertices $s, t \in V$, is there a path from s to t? We pinpoint the degree of both of these problems for both undirected and directed graphs. The proof strategy is similar in all four cases: we induct on the number of vertices n, decompose larger problems into smaller problems, and use this decomposition to analyze the highest-degree Fourier coefficient.

5.2.1 Undirected Connectivity

We begin with the undirected case. Recall that to prove that a Boolean function has maximum-possible degree, it suffices to show that its parity difference is non-zero.

Theorem 5.3 (Undirected Connectivity) *The degree of the Boolean function for undirected* CONNECTIVITY *on graphs with n vertices is $\binom{n}{2}$.*

Proof: Let f_n denote the Boolean function representing undirected CONNECTIVITY for graphs with n vertices. We prove that

$$\Phi(f_n) = (-1)^{n-1}(n-1)!, \tag{2}$$

which implies the theorem.

We proceed by induction on n. The base case $n = 1$ is trivial; the graph is always connected and never has any edges, so $\Phi(f_1) = 1$.

Suppose that our inductive hypothesis is true for all smaller values of n. Consider the first vertex of our graph on n vertices. If we begin with a connected graph and remove this vertex, we obtain a partition of the remaining $n - 1$ vertices into connected components. We can split the set S of all connected graphs into sets S_P which yield the same partition P of connected components when the first vertex is removed. By Lemma 5.2(a), $\Phi(S) = \sum_P \Phi(S_P)$.

Consider a particular partition P of $n - 1$ vertices. Each class of the partition must be connected, and the first vertex of the graph must be connected to each partition class with at least one edge. Thus, S_P can be thought of as a cross product between the set of ways to connect the first vertex to the first class, the set of ways to connect the first class, the set of ways to connect the first vertex to the second class, the set of ways to connect the second class, and so on. By Lemma 5.2(b), it suffices to compute the parity difference of each of these sets separately, and take their product.

First, we consider the set of ways to connect the first vertex to a class with k vertices. There are k possible edges, and the only forbidden choice is the one with no edges. Hence there are 2^{k-1} odd choices but only $2^{k-1} - 1$ even choices (the missing choice is even), and the parity difference of this set is -1.

By induction, the set of ways of connecting a class of size k has parity difference $(-1)^{k-1}(k-1)!$. Multiplying this with the parity difference of the set of ways that the first vertex can be connected to the class, the factor of a partition's parity difference corresponding to a class of size k is $(-1)^k(k-1)!$. The contribution of a single partition is the product of such terms, with one term per class.

Finally, we need to sum the parity differences over all partitions. Since these are all partitions of $n - 1$ vertices, they all have the same sign: $(-1)^{n-1}$. We need to show that the total magnitude is $(n - 1)!$ to complete the proof.

To see this, consider all $(n-1)!$ permutations of $n-1$ items. Think of a permutation in cycle notation, with one cycle per orbit of the permutation, and with the smallest element of the cycle written first. The cycles of a permutation yield a partition of $n-1$ vertices. A class of k vertices can be created by any of $(k-1)!$ possible cycles on these vertices. Multiplying over the classes of a partition, we see that the number of permutations that map to a partition is exactly the magnitude of the parity difference of the partition. This implies that the total parity difference magnitude is the number $(n-1)!$ of permutations, which completes the inductive hypothesis and the proof. ■

Corollary 5.4 (Lower Bound for Undirected CONNECTIVITY) *Every s–*SHUFFLE *or s–*SHUFFLE*(Σ) computation that computes the undirected* CONNECTIVITY *function uses at least* $\lceil \log_s \binom{n}{2} \rceil \approx 2\log_s n - \log_s 2$ *rounds.*

Theorem 5.5 (Undirected s-t Connectivity) *The degree of the Boolean function representing undirected* ST-CONNECTIVITY *on graphs with n vertices is $\binom{n}{2}$.*

Proof: Let f_n denote the Boolean function representing undirected ST-CONNECTIVITY for graphs with n vertices. By convention, we assume that s and t are the first and second vertices. We show that $\Phi(f_n) = (-1)^{n-1}(n-2)! \neq 0$.

It is convenient to analyze $-\Phi(f_n)$ and consider all graphs in which s and t are *not* connected. Let $S_{A,B}$ be the subset of such graphs in which the connected component of s is A and the connected component of t is B. By Lemma 5.2(a), $-\Phi(f_n)$ is the sum of the $\Phi(S_{A,B})$'s for all such A, B.

Fix A and B. Both are connected, and the remainder of the vertices of the graph can have an arbitrary subset of edges within it. We can therefore think of $S_{A,B}$ as a product of the set of ways to connect A, the set of ways to connect B, and, if there are at least two vertices outside $A \cup B$, the set of ways to pick edges in the remainder of the graph. By Lemma 5.2(b), we only need to compute the parity differences of each of these sets and take their product.

First, we observe that if $V \setminus (A \cup B)$ contains more than one vertex, then the parity difference of the last set (all subsets of edges of $V \setminus (A \cup B)$) is zero. For the rest of the proof, we focus on sets A, B whose union includes all vertices, except possibly for one.

If A has a vertices and B has b vertices, then by (2) the parity difference for the set of ways to connect A is $(-1)^{a-1}(a-1)!$ and the parity difference for the set of ways to connect B is $(-1)^{b-1}(b-1)!$.

Since $V \setminus (A \cup B)$ has zero or one vertex, there are $\frac{(n-2)!}{(a-1)!(b-1)!}$ ways to choose a set A of size a and a set B of size b (in the latter case, there are $n-2$ choices for the missing vertex, and $\binom{n-3}{a-1}$ ways of selecting the vertices that join s in A). Thus, the total contribution to the parity difference of sets A, B with sizes a, b is $(-1)^{n-2}(n-2)!$ (if $a+b=n$) or $(-1)^{n-3}(n-2)!$ (if $a+b=n-1$). There are $n-1$ ways to choose $a \geq 1$ and $b \geq 1$ so that $a+b=n$, and $n-2$ ways to choose $a \geq 1$ and $b \geq 1$ so that $a+b=n-1$. Our final parity difference (for $-\Phi$) is

$$(n-1)(-1)^{n-2}(n-2)! + (n-2)(-1)^{n-3}(n-2)! = (-1)^{n-2}(n-2)!.$$

This completes the proof. ■

Corollary 5.6 (Lower Bound for Undirected ST-CONNECTIVITY) *Every s–*SHUFFLE *or s–*SHUFFLE*(Σ) computation that computes the undirected* ST-CONNECTIVITY *function uses at least* $\lceil \log_s \binom{n}{2} \rceil \approx 2\log_s n - \log_s 2$ *rounds.*

5.2.2 Directed Connectivity

We now consider the more complicated case of directed graphs, where we ask whether the graph is *strongly* connected.

Theorem 5.7 (Directed Connectivity) *The degree of the Boolean function for directed* CONNECTIVITY *on graphs with n vertices is $2\binom{n}{2}$.*

Proof: The proof is similar but somewhat trickier than that for undirected CONNECTIVITY. Letting f_n denote the Boolean function for directed CONNECTIVITY on graphs with n vertices, we show by induction on n that $\Phi(f_n) = (n-1)!$. The base case of $n = 1$ is trivial, as the graph is always connected and $\Phi(f_1) = 1$.

For the inductive step, fix $n > 1$ and consider the first vertex of a strongly connected graph on n vertices. If we remove this vertex, then the remaining $n-1$ vertices are partitioned into strongly connected components. We can split the set S of all strongly connected graphs into sets S_P that yield the same partition P after the first vertex is removed. By Lemma 5.2(a), $\Phi(S) = \sum_P \Phi(S_P)$.

We claim that total parity difference contributions $\Phi(S_P)$ of the partitions P of the last $n-1$ vertices in which there is at least one edge between two classes is zero. To see this, order the edges that span different classes arbitrarily, and bucket the strongly connected graphs that induce P according to the first such edge (v, w) that it includes. Now fix (v, w), let v be in class A, w in class B, and let u denote the first vertex of the graph. For each graph G in this bucket that includes the edge (u, w), the graph $G \setminus \{(u, w)\}$ is also in this bucket — since all graphs in the bucket include edge (v, w) and contain a path from u to all vertices in A, $G \setminus \{(u, w)\}$ is also strongly connected. Thus, we can uniquely pair up the even and odd solutions of this bucket. Since the buckets are disjoint, we can pair up the even and odd strongly connected graphs that include at least one edge that spans two classes.

We now consider only strongly connected graphs with no edges between the classes of the partition P. Fixing P, we can view these graphs as a product of the set of directed edges between the first vertex and the first class, the set of ways to choose how the first class is strongly connected, the set of directed edges between the first vertex and the second class, the set of ways to choose how the second class is strongly connected, and so on. By Lemma 5.2(b), we can compute the parity difference of each of these sets separately, and take their product.

First, we consider the number of ways to connect the first vertex to a class with k vertices. There are k possible edges to the class, and k possible edges from the class. We count the parity differences separately and invoke Lemma 5.2(b) to compute the total parity difference by multiplying. For the k edges to the class, the only forbidden choice is the empty set. Hence there are 2^{k-1} odd choices but only $2^{k-1}-1$ even choices. The parity difference here is -1. But the same reasoning applies to the k edges from the class, so the total parity difference is 1.

By induction, the parity difference of the number of ways to choose how a class of size k is strongly connected is $(k-1)!$. If we include the number of ways that the first vertex can be connected to the class, the net contribution to the partition from a class of size k is $(k-1)!$.

Finally, we need to sum the parity difference over all partitions, which we claim is $(n-1)!$. As in the proof of Theorem 5.3, partitions with classes weighted by $(k-1)!$ correspond to permutations of $n-1$ items, and the total parity difference is $(n-1)!$. This proves the inductive step and completes the proof. ■

Corollary 5.8 (Lower Bound for Directed CONNECTIVITY) *Every s–*SHUFFLE *or s–*SHUFFLE*(Σ) computation that computes the directed* CONNECTIVITY *function uses at least $\lceil \log_s 2\binom{n}{2} \rceil \approx 2\log_s n$ rounds.*

Finally, we consider the directed ST-CONNECTIVITY problem. In contrast to the other three cases, the degree of the corresponding Boolean function is only half of the maximum possible. The proof is correspondingly more intricate, as we cannot simply prove that the parity difference of the function is non-zero.

Theorem 5.9 (Directed s-t Connectivity) *The degree of the Boolean function for directed* ST-CONNECTIVITY *on graphs with n vertices is $\binom{n}{2}$.*

Proof: Let f_n denote the Boolean function for directed ST-CONNECTIVITY on graphs with n vertices. First, we show that $deg(f_n) \geq \binom{n}{2}$. Suppose that the polynomial for directed ST-CONNECTIVITY is $p(x)$, and index x by possible edges in our graph, i.e. $x_{(i,j)}$ denotes whether edge (i,j) is in the graph. Suppose we plug in both $x_{(i,j)} = y_{(i,j)}$ for all $i < j$ and $x_{(j,i)} = y_{(i,j)}$ for all $i < j$ and simplify. In other words, we replace variables for each pair of directed edges with a single variable for an undirected edge. The polynomial now computes undirected ST-CONNECTIVITY. Its degree can only be lower than that of the original polynomial for directed ST-CONNECTIVITY, because simplifying consists only of applying the rule $x^2 = x$ and combining like terms. By Theorem 5.5, the degree for undirected ST-CONNECTIVITY is $\binom{n}{2}$, so the degree for directed ST-CONNECTIVITY is at least that.

In the rest of the proof, we show that $deg(f_n) \leq \binom{n}{2}$. We continue to use $p(x)$ to represent the polynomial for directed ST-CONNECTIVITY. We use $\mathcal{E}$ to denote the set of all possible edge sets over V. Then written as a sum of monomials, $p(x)$ looks like:

$$p(x) = \sum_{E \in \mathcal{E}} \alpha_E \prod_{e \in E} x_e, \tag{3}$$

where the α_E's are constant coefficients.

We will want to evaluate $p(x)$ for particular choices of x. We write x_E for the vector where $x_e = 1$ if and only if $e \in E$. Evaluating $p(x_E)$ then yields whether there is an s-t path using only edges in E. Notice that the only non-zero terms in (3) correspond to edge sets E' with $E' \subseteq E$, and so

$$p(x_E) = \sum_{E' \subseteq E} \alpha_{E'}.$$

Our first claim is that $\alpha_E = 0$ for any edge set $E \in \mathcal{E}$ that contains an edge $e \in E$ which is not used by any simple s-t path that only goes through edges in E. We will prove this claim by induction on the size of E.

The base case of our induction, when $|E| = 0$, is trivially true because E does not have any edges. For the inductive step, consider an edge set E with such an edge e. Let $\bar{E} = E \setminus e$. Because e is not used in any simple s-t paths, removing it does not change whether there is an s-t path. In other words, $p(x_E) = p(x_{\bar{E}})$. We can then expand the expression for $p(x_E)$, treating subsets E' of E that contain e and those that do not separately:

$$\begin{aligned} p(x_E) &= \sum_{E' \subseteq E : e \in E'} \alpha_{E'} + \sum_{E' \subseteq E : e \notin E'} \alpha_{E'} \\ &= \sum_{E' \subseteq E : e \in E'} \alpha_{E'} + p(x_{\bar{E}}) \end{aligned}$$

and $\sum_{E'\subseteq E: e\in E'} \alpha_{E'} = 0$. Our inductive hypothesis states that for every edge set E' with fewer edges than E and which contains an edge $e \in E'$ which is not used in any simple s-t path, we have that $\alpha_{E'} = 0$. Since e was not used in any simple s-t path containing only edges from E, e is also not used in any simple s-t path containing only edges from any E'. Hence $\alpha_{E'} = 0$ for every term in this summation with $E' \neq E$, which in turn implies that $\alpha_E = 0$ as well. This completes the proof of our first claim.

Our second claim is that for every edge set $E \in \mathcal{E}$ which contains both an edge $e = (i, j)$ and its reverse edge $e' = (j, i)$, we have $\alpha_E = 0$. This second claim is enough to complete the proof, since it implies that every edge set $E \in \mathcal{E}$ with a nonzero α_E has at most $\binom{n}{2}$ edges, and hence the degree of the polynomial is at most $\binom{n}{2}$. We also prove our second claim by induction on the size of E.

The base case $|E| = 0$ is again trivial. For the inductive step, consider an edge set E with such a pair of edges $e = (i, j)$ and $e' = (j, i)$.

Due to our first claim, α_E is zero if either of e or e' is not used in a simple s-t path. Hence we need only consider the case where there are simple s-t paths using both of these edges. This implies that there must be s-i, s-j, i-t, and j-t paths, none of which use e or e'. Let $E_1 = E\setminus\{e\}$, $E_2 = E\setminus\{e'\}$, and $E_3 = E\setminus\{e, e'\}$. We know that all of E, E_1, E_2, and E_3 still have s-t paths, i.e. $p(x_E) = p(x_{E_1}) = p(x_{E_2}) = p(x_{E_3}) = 1$. Similar to the proof of our first claim, we can manipulate the expression for $p(x_E)$ by considering the four different types of subsets $E' \subseteq E$ separately (depending on whether or not E' contains e, and whether or not it contains e'):

$$\begin{aligned} p(x_E) &= \sum_{\substack{E'\subseteq E:\\ e\in E',\\ e'\in E'}} \alpha_{E'} + \sum_{\substack{E'\subseteq E:\\ e\in E',\\ e'\notin E'}} \alpha_{E'} + \sum_{\substack{E'\subseteq E:\\ e\notin E',\\ e'\in E'}} \alpha_{E'} + \sum_{\substack{E'\subseteq E:\\ e\notin E',\\ e'\notin E'}} \alpha_{E'} \\ &= \sum_{\substack{E'\subseteq E:\\ e\in E'\\ e'\in E'}} \alpha_{E'} + [p(x_{E_2}) - p(x_{E_3})] + [p(x_{E_1}) - p(x_{E_3})] + [p(x_{E_3})] \end{aligned}$$

and so

$$1 = \sum_{\substack{E'\subseteq E:\\ e\in E'\\ e'\in E'}} \alpha_{E'} + [1-1] + [1-1] + [1]$$

and

$$\sum_{\substack{E'\subseteq E:\\ e\in E'\\ e'\in E'}} \alpha_{E'} = 0.$$

Our inductive hypothesis states that for every edge set E' with fewer edges than E and which contains edges $e = (i, j)$ and $e' = (j, i)$, $\alpha_{E'} = 0$. Again, this implies that every term in the summation other than α_E must be zero, and hence $\alpha_E = 0$ as well. This completes the proof of the second claim and the theorem. ■

Corollary 5.10 (Lower Bound for Directed ST-CONNECTIVITY) *Every s–SHUFFLE or s–SHUFFLE(Σ) computation that computes the directed* ST-CONNECTIVITY *function uses at least* $\lceil \log_s \binom{n}{2} \rceil \approx 2\log_s n - \log_s 2$ *rounds.*

Remark 5.11 (Graphs Represented with Adjacency Lists) Similar ideas can be used to prove degree lower bounds for connectivity problems where the input graph is represented using adjacency lists rather than an adjacency matrix. For example, consider the undirected ST-CONNECTIVITY problem for graphs with n

vertices and m edges. The input is an m-vector, with each component drawn from an alphabet of size $\binom{n}{2}$ (the possible edges). The problem is trivial if m is sufficiently close to n^2 (every graph will be connected), so assume that $1 \le m \le n^2/8$.

We prove the lower bound using only a family of 2^m different graphs. Assume that s and t are the vertices 0 and 1. The first edge is either $(0,1)$ or $(0,2)$, the second either $(1,2)$ or $(1,3)$, the third either $(0,3)$ or $(0,4)$, the fourth either $(2,3)$ or $(2,4)$, and so on. In general, each edge is drawn from a pair of the form $\{(a,b),(a,b+1)\}$, where $a < b$ and a, b have opposite parities, and pairs are considered in nondecreasing order of b, with ties broken in favor of smaller values of a. There are at least $n^2/8$ pairs of this form. Since $m \le n^2/8$, we can consider the first m such pairs and the corresponding 2^m different inputs. The key observation is that such an input fails to contain an s-t path if and only if, for each edge, the second option is chosen. As long as the second option is chosen, the graph has two connected components, one spanning the even-parity vertices considered so far (including 0) and the other the odd-parity vertices considered so far (including 1). If the first option is ever chosen, this edge connects the two components, and in particular 0 and 1. Restricted to these 2^m inputs, the ST-CONNECTIVITY function computes the OR_m function. Since there is no smaller-than-degree-m polynomial representation of the OR_m function (Corollary 4.1), there is no such representation of the ST-CONNECTIVITY function.

6 Prospects for Stronger Lower Bounds

Our lower bounds in Section 3–4 apply to general unbounded-width s–SHUFFLE computations, and by Proposition 3.4 such lower bounds cannot be larger than $\lceil \log_s n \rceil$. Realizable MapReduce-type computations translate to s–SHUFFLE computations with a reasonable number of machines, so a natural question is whether or not stronger lower bounds hold for a polynomial number of machines. For example, can we prove that there is no $O(1)$-round s–SHUFFLE computation for graph connectivity problems with $s = \sqrt{n}$ and a polynomial number of machines? In this section, we show that proving such stronger impossibility results requires proving new circuit lower bounds.

The following result, while not difficult to prove, has significant implications for the prospects of strong lower bounds for parallel computation.

Theorem 6.1 (Circuit Complexity Barrier to Stronger Lower Bounds) *Consider any model of parallel computation with the following properties:*

1. *The number of machines is polynomial in the input size n.*
2. *Computation proceeds in time steps. In each time step, a machine can read $s(n) \ge 2$ bits from the input or from previously completed computations.*
3. *Each machine has enough power to evaluate a Boolean circuit with size at most $s(n)$ and depth at most $\log_2 s(n)$ in one time step.*

If some problem in P cannot be solved in $O(\log_{s(n)} n)$ time steps in such a model, then $NC^1 \subsetneq P$.

Property (1) states that an arbitrarily large polynomial number of machines is allowed. Property (2) asks for a complete communication graph and concurrent reads, and property (3) only requires fairly weak computing power per machine. s–SHUFFLE computations with a polynomial number of machines are certainly one example of a model that satisfies these assumptions. The point of Theorem 6.1 is that, more generally,

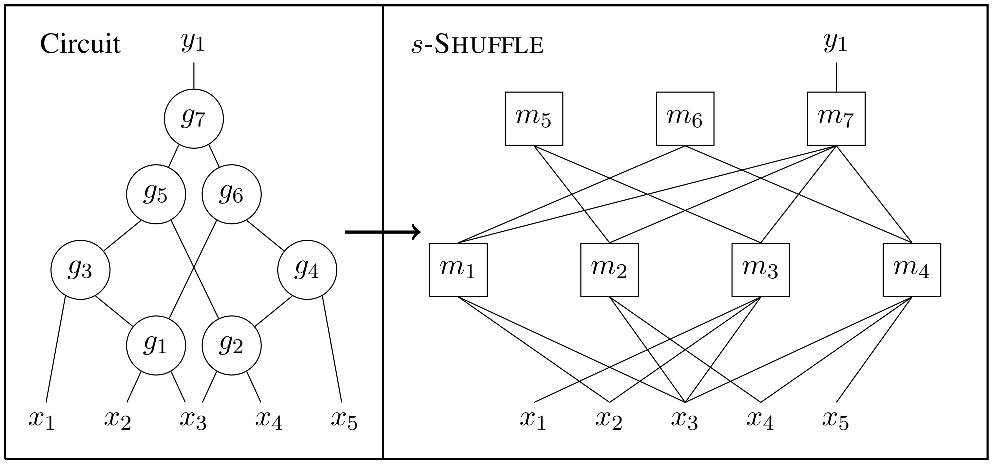

Figure 2: A depth-four circuit and its corresponding s-SHUFFLE with $s = 4$ that runs in two rounds. The number on a machine matches the number of the gate it computes.

the *only possible way* of proving lower bounds stronger than $\Theta(\log_s n)$ for any model of parallel computation is to either (i) prove a notoriously difficult circuit lower bound or (ii) restrict the model so much that one of the hypotheses in the theorem is not satisfied. In particular, Theorem 6.1 leaves open the possibility of proving strong lower bounds for restricted classes of algorithms (where the third property may not hold), and some of the previous work reviewed in Section 1.2 is of this type.

Proof of Theorem 6.1: We simply need to show that every model with these three properties can efficiently simulate NC^1 circuits. Fix some such model of parallel computation. Recall that NC^1 is the family of problems that can be computed by a logspace-uniform circuit family C_n of fan-in 2 with $poly(n)$ gates and $O(\log n)$ depth (where n denotes the number of inputs). We will transform each circuit into a parallel computation that uses only $O(\log_{s(n)} n)$ time steps (see Figure 2 for an example). This will show that no NC^1 family computes our hypothesized problem, and hence $NC^1 \subsetneq P$, as desired.

By the first assumed property, we can associate each gate of the circuit with a machine, whose sole responsibility is to compute the output of the gate. Each time step of the parallel computation will correspond to $\log_2 s(n)$ layers of the circuit. For example, in the first time step, we will have a machine for every gate at depth at most $\log_2 s(n)$. Since the circuit has fan-in 2, computing the output of this gate requires reading at most $s(n)$ input bits and evaluating a Boolean circuit with depth at most $\log_2 s(n)$ and size at most $s(n)$. By the second and third assumed properties, the corresponding machine can perform this computation in the first time step of the parallel computation. In general, in the i^{th} time step, we have a machine for every gate at depth between $(i-1)\log_2 s(n)$ and $i \log_2 s(n)$. Again, the output of each of these gates can be computed by reading at most $s(n)$ bits (from the input or the outputs of gates corresponding to machines in previous time steps) and evaluating a depth-$\log_2 s(n)$ and size-$s(n)$ Boolean circuit. We reach the output gates and complete the computation in $O(\frac{\log n}{\log s(n)}) = O(\log_{s(n)} n)$ time steps of parallel computation, as desired. ■

The simulation argument in the proof of Theorem 6.1 is quite versatile. For example, an analogous statement holds in the uniform setting. Since NC^1 is defined to be logspace-uniform, the lower bound barrier holds even if the model of parallel computation is restricted to logspace-uniform computations at each machine. Also, the simulation in the proof of Theorem 6.1 only blows up the width by a factor of $\log_2 s(n)$ from the circuit to parallel computation. Hence, given a round lower bound on a width-restricted model of parallel computation, we would get a depth lower bound on a width-restricted (less so by a factor $\log_2 s(n)$) version of NC, which would still be a significant result in circuit complexity. Similarly, the simulation results in the same number of machines as there were gates, so a round lower bound for a machine-restricted model of parallel computation would give a depth lower bound for a size-restricted version of NC.

Acknowledgments

We thank Noam Nisan for pointing us to several relevant references, and the SPAA and JACM referees for several helpful comments.

References

[1] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *Proceedings of the VLDB Enfowment (PVLDB)*, 6(4):277–288, 2013.

[2] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[3] A. Andoni, A. Nikolov, K. Onak, and G. Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, pages 574–583, 2014.

[4] M. Bateni, A. Bhaskara, S. Lattanzi, and V. S. Mirrokni. Distributed balanced clustering via mapping coresets. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 2591–2599, 2014.

[5] R. Beals, H. Buhrman, R. Cleve, M. Mosca, and R. De Wolf. Quantum lower bounds by polynomials. *Journal of the ACM*, 48(4):778–797, 2001.

[6] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 273–284, 2013.

[7] G. Bilardi, M. Scquizzato, and F. Silvestri. A lower bound technique for communication on BSP with application to the FFT. In *European Conference on Parallel Processing*, pages 676–687, 2012.

[8] H. Buhrman and R. De Wolf. Complexity measures and decision tree complexity: a survey. *Theoretical Computer Science*, 288(1):21–43, 2002.

[9] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, 1986.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150, 2004.

[11] M. Dietzfelbinger, M. Kutyłowski, and R. Reischuk. Exact lower time bounds for computing Boolean functions on CREW PRAMs. *Journal of Computer and System Sciences*, 48(2):231–254, 1994.

[12] R. O. Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014.

[13] A. Drucker, F. Kuhn, and R. Oshman. On the power of the congested clique model. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 367–376, 2014.

[14] A. Ene, S. Im, and B. Moseley. Fast clustering using MapReduce. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 681–689, 2011.

[15] J. Feldman, S. Muthukrishnan, A. Sidiropoulos, C. Stein, and Z. Svitkina. On distributing symmetric streaming computations. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 710–719, 2008.

[16] B. Fish, J. Kun, Á. D. Lelkes, L. Reyzin, and G. Turán. On the computational complexity of MapReduce. In *Proceedings of the 29th International Symposium on Disributed Computing (DISC)*, pages 1–15, 2015.

[17] A. Goel and K. Munagala. Complexity measures for Map-Reduce, and comparison to parallel computing. arXiv:1211.6526, 2012.

[18] M. T. Goodrich. Communication-efficient parallel sorting. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC)*, pages 247–256, 1996.

[19] M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, searching, and simulation in the MapReduce framework. In *Proceedings of the 22nd International Symposium on Algorithms and Computation (ISAAC)*, pages 374–383, 2011.

[20] J. W. Hegeman and S. V. Pemmaraju. Lessons from the congested clique applied to MapReduce. *Theoretical Computer Science*, 608:268–281, 2015.

[21] P. Hrubes and A. Rao. Circuits with medium fan-in. In *Proceedings of the 30th Computational Complexity Conference (CCC)*, pages 381–391, 2015.

[22] R. Jacob, T. Lieber, and N. Sitchinava. On the complexity of list ranking in the parallel external memory model. In *Proceedings of the 39th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 384–395, 2014.

[23] J. Kahn, M. Saks, and D. Sturtevant. A topological approach to evasiveness. *Combinatorica*, 4(4):297–306, 1984.

[24] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

[25] H. Klauck, D. Nanongkai, G. Pandurangan, and P. Robinson. Distributed computation of large-scale graph problems. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 391–410, 2015.

[26] D. J. Kleitman and D. J. Kwiatkowski. Further results on the Aanderaa-Rosenberg conjecture. *Journal of Combinatorial Theory, Series B*, 28(1):85–95, 1980.

[27] T. Korneffel and E. Triesch. An asymptotic bound for the complexity of monotone graph properties. *Combinatorica*, 30(6):735–743, 2010.

[28] R. Kumar, B. Moseley, S. Vassilvitskii, and A. Vattani. Fast greedy algorithms in MapReduce and streaming. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 1–10, 2013.

[29] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 85–94, 2011.

[30] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014. Second Edition.

[31] V. Mirrokni and M. Zadimoghaddam. Randomized composable core-sets for distributed submodular maximization. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, 2015.

[32] N. Nisan. CREW PRAMs and decision trees. *SIAM Journal on Computing*, 20(6):999–1007, 1991.

[33] N. Nisan and M. Szegedy. On the degree of Boolean functions as real polynomials. *Computational Complexity*, 4(4):301–313, 1994.

[34] D. Peleg. *Distributed Computing*. SIAM, 2000.

[35] A. Pietracaprina, G. Pucci, M. Riondato, F. Silvestri, and E. Upfal. Space-round tradeoffs for MapReduce computations. In *Proceedings of the International Conference on Supercomputing*, pages 235–244, 2012.

[36] A. A. Razborov. Lower bounds on the size of bounded depth circuits over a complete basis with logical addition. *Mathematical Notes of the Academy of Sciences of the USSR*, 41(4):333–338, 1987.

[37] R. L. Rivest and J. Vuillemin. A generalization and proof of the Aanderaa-Rosenberg conjecture. In *Proceedings of the 7th Annual ACM Symposium on Theory of Computing (STOC)*, pages 6–11, 1975.

[38] A. L. Rosenberg. On the time required to recognize properties of graphs: A problem. *SIGACT News*, 5(4):15–16, 1973.

[39] R. Scheidweiler and E. Triesch. A lower bound for the complexity of monotone graph properties. *SIAM Journal on Discrete Mathematics*, 27(1):257–265, 2013.

[40] R. Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 77–82, 1987.

[41] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[42] H. Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag, 1999.

[43] D. P. Woodruff and Q. Zhang. When distributed computation is communication expensive. In *Proceedings of the 27th International Symposium on Disributed Computing (DISC)*, pages 16–30, 2013.

[44] A. C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981.