**COSC 544 Probabilistic Proof Systems** 9/25/17

# The GKR Protocol and Its Efficient Implementation

*Lecturer: Justin Thaler*

## 1 Motivation

The goal in the lecture on LFKN's #SAT protocol was to develop an interactive proof for an intractable problem (such as #SAT [LFKN92] or TQBF [Sha92]), in which the verifier ran in polynomial time. The perspective taken in this lecture is different: it acknowledges that there are no "real world" entities that can act as the prover in the #SAT and TQBF protocols of earlier lectures, since real world entities cannot solve large instances of **PSPACE**-complete problems in the worst case. We would really like a "scaled down" version of the TQBF protocol of [LFKN92, Sha92], one that is useful for problems that can be solved in the real world, such as problems in the complexity classes **P**, or **NC** (capturing problems solvable by efficient parallel algorithms), or even **L** (capturing problems solvable in logarithmic space).

But what is the point of developing verification protocols for such easy problems? Can't the verifier just ignore the prover and solve the problem without help? The answer is that this lecture will describe protocols in which the verifier runs much faster than would be possible without a prover. Specifically, $\mathcal{V}$ will run linear time, doing little more than just reading the input.

Meanwhile, we will require that the prover not do much more than solve the problem of interest. Ideally, if the problem is solvable by a Random Access Machine or Turing Machine in time $T$ and space $s$, we want the prover to run in time $O(T)$ and space $O(s)$, or as close to it as possible. At a minimum, $\mathcal{P}$ should run in polynomial time.

Can the TQBF and #SAT protocols of prior lectures be scaled down to yield protocols where the verifier runs in (quasi-)linear time for a "low" complexity class like **L**? It turns out that it can, but the prover is not efficient.

Recall that in the TQBF protocol, $\mathcal{V}$ ran in time $O(S)$, and $\mathcal{P}$ ran in time $O(S \cdot 2^N)$, when applied to a Boolean formula $\phi$ of size $S$ over $N$ variables. In principle, this yields an interactive proof for any problem solvable in space $s$: given an input $\mathbf{x} \in \{0,1\}^n$, $\mathcal{V}$ first transforms $\mathbf{x}$ to an instance $\phi$ of TQBF (see, e.g., [AB09, Chapter 4] for a lucid exposition of this transformation, which is reminiscent of Savitch's Theorem), and then applies the interactive proof for TQBF to $\phi$.

However, the transformation yields a TQBF instance $\phi$ over $N = O(s \cdot \log T)$ variables when applied to a problem solvable in time $T$ and space $s$. This results in a prover that runs in time in time $2^{O(s \cdot \log T)}$. This is superpolynomial (i.e., $n^{\Theta(\log n)}$), even if $s = O(\log n)$ and $t = \text{poly}(n)$. Until 2007, this was the state of the art in interactive proofs.

## 2 The GKR Protocol and Its Costs

Goldwasser, Kalai, and Rothblum [GKR08] described a remarkable interactive proof protocol that does achieve many of the goals set forth above. The protocol is best presented in terms of the (arithmetic) *circuit evaluation* problem. In this problem, $\mathcal{V}$ and $\mathcal{P}$ first agree on a (logspace uniform) arithmetic circuit $\mathcal{C}$ of fan-in 2 over a finite field $\mathbb{F}$, and the goal is to compute the value of the output gate(s) of $\mathcal{C}$. A logspace uniform circuit is one that possess a succinct implicit description, in the sense that there is a log-space algorithm that

| Communication | Rounds | $\mathcal{V}$ time | $\mathcal{P}$ time |
|---|---|---|---|
| $d \cdot \text{polylog}(S)$ field elements | $d \cdot \text{polylog}(S)$ | $O(n + d \cdot \text{polylog}(S))$ | $\text{poly}(S)$ |

Table 1: Costs of the GKR protocol when applied to any log-space uniform arithmetic circuit $\mathcal{C}$ of size $S$ and depth $d$ over $n$ variables. Section 4.1 describes methods from [CMT12] for reducing $\mathcal{P}$'s runtime to $O(S \log S)$, and reducing the $\text{polylog}(S)$ terms in the remaining costs to $O(\log S)$.

takes as input any number $i \in \{0, \ldots, S-1\}$, and outputs the identities of the in-neighbors of the $i$th gate in $\mathcal{C}$.

Letting $S$ denote the size (i.e., number of gates) of $\mathcal{C}$ and $n$ the number of variables, the key feature of the GKR protocol is that the prover runs in time $\text{poly}(S)$ (we will see that $\mathcal{P}$'s time can even be made nearly *linear* in $S$ [CMT12, Tha13]). If $S = 2^{o(n)}$, then this is much better than the TQBF protocol that we saw in an earlier lecture, where the prover required time exponential in the number of variables over which the TQBF instance was defined.

Moreover, the costs to the verifier in the GKR protocol grow linearly with the *depth* $d$ of $\mathcal{C}$, and only logarithmically with $S$. Crucially, this means that $\mathcal{V}$ can run in time *sub-linear* in the size of the circuit $S$. At first glance, this might seem impossible – how can the verifier make sure the prover correctly evaluated $\mathcal{C}$ if the verifier never even "looks" at all of $\mathcal{C}$? The answer is that $\mathcal{C}$ was assumed to have a succinct implicit description in the sense of being log-space uniform. This enables $\mathcal{V}$ to "understand" the structure of $\mathcal{C}$ without ever having to look at every gate individually.

**Application: An Interactive Proof for Parallel Algorithms.** The complexity class **NC** consists of languages solvable by parallel algorithms in time $\text{polylog}(n)$ and total work $\text{poly}(n)$. Any problem in **NC** can solved by a (log-space uniform) arithmetic circuit $\mathcal{C}$ of polynomial size and polylogarithmic depth. Applying the GKR protocol to $\mathcal{C}$ yields a polynomial time prover and a linear time verifier.

**Application: An Interactive Proof for Space-Bounded Turing Machines.** As discussed above, the TQBF protocol from earlier lectures can be used to give a prover running in time $2^{O(\log s \cdot \log T)}$ and a verifier running in time $O(n \cdot \text{poly}(s))$ when applied to a language solvable in time $T$ and space $s$. In particular, $\mathcal{P}$'s runtime is superpolynomial even if $s = O(\log n)$.

The GKR protocol provides a quantitative improvement on these costs. Specifically, any language $\mathcal{L}$ solvable in time space $s$ and time $T$ is computed by an $O(\log s)$-space uniform circuit $\mathcal{C}$ of fan-in 2 and depth $d = O(s \cdot \log T)$ and size $S = \text{poly}(2^s)$. Roughly speaking, $\mathcal{C}$ works by taking a Turing Machine or Random Access Machine $M$ computing $\mathcal{L}$, and repeatedly squaring the adjacency matrix of $M$ in order to determine whether there is a path from the start configuration of $M$ to the accepting configuration of $M$. Only $O(\log T)$ invocations of MATMULT are required to determine if a path of length $T$ exists, and each invocation can be computed in depth $O(s)$, resulting in the $O(s \cdot \log T)$ bound on the depth of $\mathcal{C}$.

Applying the GKR protocol to $\mathcal{C}$, the prover runs in time $\text{poly}(S) = \text{poly}(2^s)$, which is polynomial if $s = O(\log n)$. The verifier runs in time $O(n + d \cdot \log S) = O(n + \text{poly}(s, \log T))$. This is $O(n)$ time if $s$ is subpolynomial in $n$ and $T$ is subexponential in $n$.

Notice the above also provides an alternate proof that **PSPACE** $\subseteq$ **IP**, as the verifier's runtime is $\text{poly}(n)$ as long as $s = \text{poly}(n)$.

# 3 Protocol Overview

As described above, $\mathcal{P}$ and $\mathcal{V}$ first agree on an arithmetic circuit $\mathcal{C}$ of fan-in 2 over a finite field $\mathbb{F}$ computing the function of interest. $\mathcal{C}$ is assumed to be in layered form, meaning that the circuit can be decomposed into layers, and wires only connect gates in adjacent layers (if $\mathcal{C}$ is not layered it can easily be transformed into a layered circuit $\mathcal{C}'$ with a small blowup in size). Suppose that $\mathcal{C}$ has depth $d$, and number the layers from 1 to $d$ with layer $d$ referring to the input layer, and layer 1 referring to the output layer.

In the first message, $\mathcal{P}$ tells $\mathcal{V}$ the (claimed) output(s) of the circuit. The protocol then works its way in iterations towards the input layer, with one iteration devoted to each layer. We will describe the gates in $\mathcal{C}$ as having values: the value of an addition (multiplication) gate is set to be the sum (product) of its in-neighbors. The purpose of iteration $i$ is to reduce a claim about the values of the gates at layer $i$ to a claim about the values of the gates at layer $i+1$, in the sense that it is safe for $\mathcal{V}$ to assume that the first claim is true as long as the second claim is true. This reduction is accomplished by applying the sum-check protocol.

More concretely, the GKR protocol starts with a claim about the values of the output gates of the circuit, but $\mathcal{V}$ cannot check this claim without evaluating the circuit herself, which is precisely what she wants to avoid. So the first iteration uses a sum-check protocol to reduce this claim about the outputs of the circuit to a claim about the gate values at layer 2 (more specifically, to a claim about an evaluation of the multilinear extension of the gate values at layer 2). Once again, $\mathcal{V}$ cannot check this claim herself, so the second iteration uses another sum-check protocol to reduce the latter claim to a claim about the gate values at layer 3, and so on. Eventually, $\mathcal{V}$ is left with a claim about the inputs to the circuit, and $\mathcal{V}$ can check this claim without any help. This outline is depicted in Figures 1-4.

# 4 Protocol Details

**Notation.** Suppose we are given a layered arithmetic circuit $\mathcal{C}$ of size $S$, depth $d$, and fan-in two. Number the layers from 0 to $d$, with 0 being the output layer and $d$ being the input layer. Let $S_i$ denote the number of gates at layer $i$ of the circuit $\mathcal{C}$. Assume $S_i$ is a power of 2 and let $S_i = 2^{s_i}$. The GKR protocol makes use of several functions, each of which encodes certain information about the circuit.

Number the gates at layer $i$ from 0 to $S_i - 1$, and let $W_i : \{0,1\}^{s_i} \to \mathbb{F}$ denote the function that takes as input a binary gate label, and outputs the corresponding gate's value at layer $i$. As usual, let $\widetilde{W}_i$ denote the multilinear extension of $W_i$.

The GKR protocol also makes use of the notion of a "wiring predicate" that encodes which pairs of wires from layer $i+1$ are connected to a given gate at layer $i$ in $\mathcal{C}$. Let $\text{in}_{1,i}, \text{in}_{2,i} \colon \{0,1\}^{s_i} \to \{0,1\}^{s_{i+1}}$ denote the functions that take as input the label $\mathbf{a}$ of a gate at layer $i$ of $\mathcal{C}$, and respectively output the label of the first and second in-neighbor of gate $\mathbf{a}$. So, for example, if gate $\mathbf{a}$ at layer $i$ computes the sum of gates $\mathbf{b}$ and $\mathbf{c}$ at layer $i+1$, then $\text{in}_{1,i}(\mathbf{a}) = \mathbf{b}$ and $\text{in}_{2,i}(\mathbf{a}) = \mathbf{c}$.

Define two functions, $\text{add}_i$ and $\text{mult}_i$, mapping $\{0,1\}^{s_i+2s_{i+1}}$ to $\{0,1\}$, which together constitute the wiring predicate of layer $i$ of $C$. Specifically, these functions take as input three gate labels $(\mathbf{a},\mathbf{b},\mathbf{c})$, and return 1 iff $(\mathbf{b},\mathbf{c}) = (\text{in}_{1,i}(\mathbf{a}), \text{in}_{2,i}(\mathbf{a}))$ and gate $\mathbf{a}$ is an addition (respectively, multiplication) gate. As usual, let $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ denote the multilinear extensions of $\text{add}_i$ and $\text{mult}_i$.

**Detailed Description.** The GKR protocol consists of $d$ iterations, one for each layer of the circuit. Each iteration $i$ starts with $\mathcal{P}$ claiming a value for $\widetilde{W}_i(\mathbf{r}_i)$ for some point in $\mathbf{r}_i \in \mathbb{F}^{s_i}$.

At the start of the first iteration, this claim is derived from the claimed outputs of the circuit. Specifically, if there are $S_0 = 2^{s_0}$ outputs of $\mathcal{C}$, let $D \colon \{0,1\}^{s_0} \to \mathbb{F}$ denote the function that maps the label of an output gate

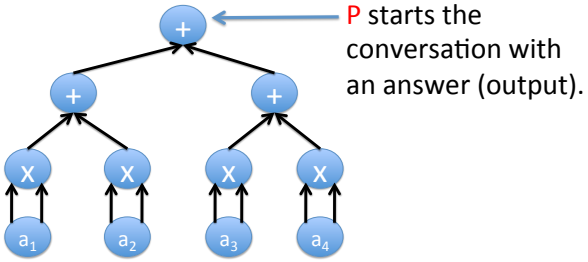


Figure 1: Start of GKR Protocol.

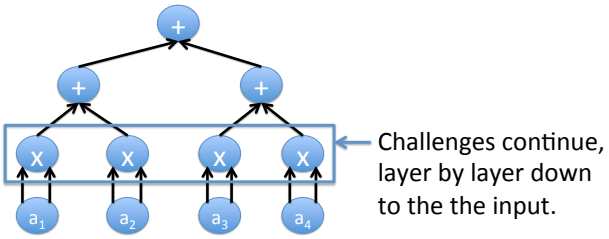


Figure 3: In general, iteration $i$ reduces a claim about the MLE of gate values at layer $i$, to a claim about the MLE of gate values at layer $i+1$.

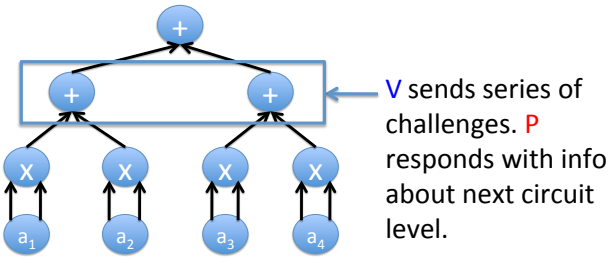


Figure 2: Iteration 1 reduces a claim about the output of $\mathcal{C}$ to one about the MLE of the gate values in the previous layer.

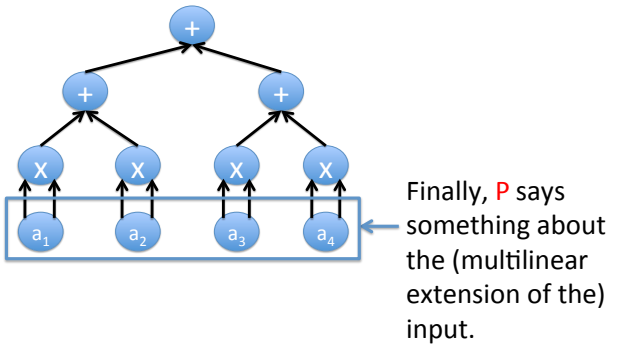


Figure 4: In the final iteration, $\mathcal{P}$ makes a claim about the MLE of the input. $\mathcal{V}$ can check this claim without help, since $\mathcal{V}$ sees the input explicitly.

to the claimed value of that output. Then the verifier can pick a random point $\mathbf{r}_0 \in \mathbb{F}^{s_0}$ at random, and evaluate $\tilde{D}(\mathbf{r}_0)$ in time $O(S_0)$ using Lemma 1.8 of Lecture 4. By the Schwartz-Zippel lemma, if $\tilde{D}(\mathbf{r}_0) = \widetilde{W}_0(\mathbf{r}_0)$ (i.e., if the multilinear extension of the claimed outputs equals the multilinear extension of the correct outputs when evaluated at a randomly chosen point), then it is safe for the verifier to believe that $\tilde{D} = \tilde{W}_0$, and hence that all of the claimed outputs are correct. Unfortunately, the verifier cannot evaluate $\widetilde{W}_0(\mathbf{r}_0)$ without help from the prover.

The purpose of iteration $i$ is to reduce the claim about the value of $\widetilde{W}_i(\mathbf{r}_i)$ to a claim about $\widetilde{W}_{i+1}(\mathbf{r}_{i+1})$ for some $\mathbf{r}_{i+1} \in \mathbb{F}^{s_{i+1}}$, in the sense that it is safe for $\mathcal{V}$ to assume that the first claim is true as long as the second claim is true. To accomplish this, the iteration applies the sum-check protocol to a specific polynomial derived from $\widetilde{W}_{i+1}$, $\widetilde{\text{add}}_i$, and $\widetilde{\text{mult}}_i$. Our description of the protocol actually makes use of a simplification due to Thaler [Tha15].

**Applying the Sum-Check Protocol.** The GKR protocol exploits an ingenious explicit expression for $\widetilde{W}_i(\mathbf{r}_i)$, captured in the following lemma.

**Lemma 4.1.**

$$\tilde{W}_i(\mathbf{z}) = \sum_{\mathbf{b},\mathbf{c} \in \{0,1\}^{s_{i+1}}} \widetilde{\text{add}}_i(\mathbf{z},\mathbf{b},\mathbf{c}) \left(\tilde{W}_i(\mathbf{b}) + \tilde{W}_i(\mathbf{c})\right) + \widetilde{\text{mult}}_i(\mathbf{z},\mathbf{b},\mathbf{c}) \left(\tilde{W}_i(\mathbf{b}) \cdot \tilde{W}_i(\mathbf{c})\right) \tag{1}$$

*Proof.* It is easy to check that the right hand side is a multilinear polynomial in the entries of $\mathbf{z}$, since $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ are multilinear polynomials. (Note that, just as in the matrix multiplication protocol of the preceding lecture, the function being summed over is *quadratic* in the entries of $\mathbf{b}$ and $\mathbf{c}$, but this quadratic-ness is "summed away", leaving a multilinear polynomial only in the variables of $\mathbf{z}$).

Since the multilinear extension of a function with domain $\{0,1\}^{s_i}$ is unique, it suffices to check that the left hand side and right hand side of the expression in the lemma agree for all $\mathbf{a} \in \{0,1\}^{s_i}$. To this end, fix any $\mathbf{a} \in \{0,1\}^{s_i}$, and suppose that gate $\mathbf{a}$ in layer $i$ of $\mathcal{C}$ is an addition gate (the case where gate $\mathbf{a}$ is a multiplication gate is similar). Since each gate $\mathbf{a}$ at layer $i$ has two unique in-neighbors, namely $\text{in}_1(\mathbf{a})$ and $\text{in}_2(\mathbf{a})$;

$$\text{add}_i(\mathbf{a},\mathbf{b},\mathbf{c}) = \begin{cases} 1 & \text{if } (\mathbf{b},\mathbf{c}) = (\text{in}_1(\mathbf{a}), \text{in}_2(\mathbf{a})) \\ 0 & \text{otherwise} \end{cases}$$

and $\text{mult}_i(\mathbf{a},\mathbf{b},\mathbf{c}) = 0$ for all $\mathbf{b},\mathbf{c} \in \{0,1\}^{s_{i+1}}$.

Hence, since $\widetilde{\text{add}}_i$, $\widetilde{\text{mult}}_i$, $\widetilde{W}_{i+1}$, and $\widetilde{W}_i$ extend $\text{add}_i$ and $\text{mult}_i$, $W_{i+1}$, and $W_i$ respectively,

$$\begin{aligned} &\sum_{\mathbf{b},\mathbf{c} \in \{0,1\}^{s_{i+1}}} \widetilde{\text{add}}_i(\mathbf{a},\mathbf{b},\mathbf{c}) \left(\tilde{W}_{i+1}(\mathbf{b}) + \tilde{W}_{i+1}(\mathbf{c})\right) + \widetilde{\text{mult}}_i(\mathbf{a},\mathbf{b},\mathbf{c}) \left(\tilde{W}_{i+1}(\mathbf{b}) \cdot \tilde{W}_{i+1}(\mathbf{c})\right) \\ &= \widetilde{W}_{i+1}(\text{in}_1(\mathbf{a})) + \widetilde{W}(\text{in}_2(\mathbf{a})) = W_{i+1}(\text{in}_1(\mathbf{a})) + W_{i+1}(\text{in}_2(\mathbf{a})) = W_i(\mathbf{a}) = \tilde{W}_i(\mathbf{a}). \end{aligned}$$

□

**Remark 1.** Lemma 4.1 is actually valid using any extensions of $\text{add}_i$ and $\text{mult}_i$ that are multilinear in the first $s_i$ variables.

**Remark 2.** Goldwasser, Kalai, and Rothblum [GKR08] actually use a slightly more complicated expression for $\tilde{W}_i(\mathbf{a})$ than the one in Lemma 4.1. Their expression allowed them to use even more general extensions of $\text{add}_i$ and $\text{mult}_i$ (in particular, their extensions do not have to be multilinear in the first $s_i$ variables).

However, our use of the multilinear extensions $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ turns out to be critical to achieving a prover runtime that is nearly *linear* in the circuit size $S$ [CMT12, Tha13], rather than a much larger polynomial in $S$ as achieved by [GKR08] (cf. Section 4.1 for details).

Therefore, in order to check the prover's claim about $\widetilde{W}_i(\mathbf{r}_i)$, the verifier applies the sum-check protocol to the polynomial

$$f^{(i)}_{\mathbf{r}_i}(\mathbf{b},\mathbf{c}) = \widetilde{\text{add}}_i(\mathbf{r}_i,\mathbf{b},\mathbf{c})\left(\tilde{W}_{i+1}(\mathbf{b})+\tilde{W}_{i+1}(\mathbf{c})\right) + \widetilde{\text{mult}}_i(\mathbf{r}_i,\mathbf{b},\mathbf{c})\left(\tilde{W}_{i+1}(\mathbf{b})\cdot\tilde{W}_{i+1}(\mathbf{c})\right). \tag{2}$$

There remains the issue that $\mathcal{V}$ can only execute the final check in the sum-check protocol if she can evaluate the polynomial $f^{(i)}_{\mathbf{r}_i}$ at a random point $\omega = (\omega_1,\ldots,\omega_{2s_{i+1}})$. This is handled as follows.

Let $\mathbf{b}^* \in \mathbb{F}^{s_{i+1}}$ be the first $s_{i+1}$ entries of $\omega$, and $\mathbf{c}^* \in \mathbb{F}^{s_{i+1}}$ the last $s_{i+1}$ entries. Note that $\mathbf{b}^*$, and $\mathbf{c}^*$ may have non-Boolean entries. Evaluating $f^{(i)}_{\mathbf{r}_i}(\mathbf{b}^*,\mathbf{c}^*)$ requires evaluating $\widetilde{\text{add}}_i(\mathbf{r}_i,\mathbf{b}^*,\mathbf{c}^*)$, $\widetilde{\text{mult}}_i(\mathbf{r}_i,\mathbf{b}^*,\mathbf{c}^*)$, $\widetilde{W}_{i+1}(\mathbf{b}^*)$, and $\widetilde{W}_{i+1}(\mathbf{c}^*)$.

For many circuits, particularly those whose wiring pattern display repeated structure, $\mathcal{V}$ can evaluate $\widetilde{\text{add}}_i(\mathbf{r}_i,\mathbf{b}^*,\mathbf{c}^*)$ and $\widetilde{\text{mult}}_i(\mathbf{r}_i,\mathbf{b}^*,\mathbf{c}^*)$ on her own in $\text{poly}(s_i,s_{i+1})$ time as well. For now, assume that $\mathcal{V}$ can indeed perform this evaluation in $\text{poly}(s_i,s_{i+1})$ time, but this issue will be discussed further in Section 4.2.

$\mathcal{V}$ cannot however evaluate $\widetilde{W}_{i+1}(\mathbf{b}^*)$, and $\widetilde{W}_{i+1}(\mathbf{c}^*)$ on her own without evaluating the circuit. Instead, $\mathcal{V}$ asks $\mathcal{P}$ to simply provide these two values, and uses iteration $i+1$ to *verify* that these values are as claimed. However, one complication remains: the precondition for iteration $i+1$ is that $\mathcal{P}$ claims a value for $\widetilde{W}_{i+1}(\mathbf{r}_{i+1})$ for a single point $\mathbf{r}_{i+1} \in \mathbb{F}^{s_{i+1}}$. So $\mathcal{V}$ needs to reduce verifying both $\widetilde{W}_{i+1}(\mathbf{b}^*)$ and $\widetilde{W}_{i+1}(\mathbf{c}^*)$ to verifying $\widetilde{W}_{i+1}(\mathbf{r}_{i+1})$ at a single point $\mathbf{r}_{i+1} \in \mathbb{F}^{s_{i+1}}$, in the sense that it is safe for $\mathcal{V}$ to accept the claimed values of $\widetilde{W}_{i+1}(\mathbf{b}^*)$ and $\widetilde{W}_{i+1}(\mathbf{c}^*)$ as long as the value of $\widetilde{W}_{i+1}(\mathbf{r}_{i+1})$ is as claimed. This is done as follows.

**Reducing to Verification of a Single Point.** Let $\ell : \mathbb{F} \to \mathbb{F}^{s_{i+1}}$ be some canonical line passing through $\mathbf{b}^*$ and $\mathbf{c}^*$. For example, we can let $\ell$ be the unique line such that $\ell(0) = \mathbf{b}^*$ and $\ell(1) = \mathbf{c}^*$. $\mathcal{P}$ sends a univariate polynomial $q$ of degree at most $s_{i+1}$ that is claimed to be $\widetilde{W}_{i+1} \circ \ell$, the restriction of $\widetilde{W}_{i+1}$ to the line $\ell$. $\mathcal{V}$ checks that $q(0) = \mathbf{b}^*$ and $q(1) = \mathbf{c}^*$ (rejecting if this is not the case), picks a random point $r^* \in \mathbb{F}$, and asks $\mathcal{P}$ to prove that $\widetilde{W}_{i+1}(\ell(r^*)) = q(r^*)$. By the Schwartz-Zippel Lemma (even its simple special case for univariate polynomials), as long as $\mathcal{V}$ is convinced that $\widetilde{W}_{i+1}(\ell(r^*)) = q(r^*)$, it is safe for $\mathcal{V}$ to believe that $q$ does in fact equal $\widetilde{W}_{i+1} \circ \ell$, and hence that the values of $\widetilde{W}_{i+1}(\mathbf{b}^*)$ and $\widetilde{W}_{i+1}(\mathbf{c}^*)$ are as claimed by $\mathcal{P}$. This completes iteration $i$; $\mathcal{P}$ and $\mathcal{V}$ then move on to the iteration for layer $i+1$ of the circuit, whose purpose is to verify that $\widetilde{W}_{i+1}(\mathbf{r}_{i+1})$ has the claimed value, where $\mathbf{r}_{i+1} := \ell(r^*)$.

**The Final Iteration.** Finally, at the final iteration $d$, $\mathcal{V}$ must evaluate $\widetilde{W}_d(\mathbf{r}_d)$ on her own. But the vector of gate values at layer $d$ of $\mathcal{C}$ is simply the input $\mathbf{x}$ to $\mathcal{C}$. By Lemma 1.8 from Lecture 4, $\mathcal{V}$ can compute $\widetilde{W}_d(\mathbf{r}_d)$ on her own in $O(n)$ time.

## 4.1 Discussion of Costs

**$\mathcal{V}$'s runtime.** Observe that the polynomial $f^{(i)}_{\mathbf{r}_i}$ defined in Equation (2) is an $(2s_{i+1})$-variate polynomial of degree at most 2 in each variable, and so the invocation of the sum-check protocol at iteration $i$ requires $2s_{i+1}$ rounds, with three field elements transmitted per round. Thus, the total communication cost is $O(d \log S)$ field elements. The time cost to $\mathcal{V}$ is $O(n + d\log S + t)$, where $t$ is the amount of time required for $\mathcal{V}$ to evaluate $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ at a random input, for each layer $i$ of $\mathcal{C}$. Here the $n$ term is due to the time required to evaluate $\widetilde{W}_d(\mathbf{r}_d)$, and the $d\log S$ term is the time required for $\mathcal{V}$ to send messages to $\mathcal{P}$ and process and check the messages from $\mathcal{P}$. For now, let us assume that $t$ is a low-order cost, so that $\mathcal{V}$ runs in total time $O(n + d\log S)$; we discuss this issue further in Section 4.2.

**$\mathcal{P}$'s runtime.** Analogously to the MATMULT protocol of the previous lecture, we give two increasingly sophisticated implementations of the prover when the sum-check protocol is applied to the polynomial $f^{(i)}_{\mathbf{r}_i}$.

***Method 1***: $f^{(i)}_{\mathbf{r}_i}$ is a $v$-variate polynomial for $v = 2s_{i+1}$. As in the analysis of Method 1 for implementing the prover in the matrix multiplication protocol from the previous lecture, $\mathcal{P}$ can compute the prescribed method in round $j$ by evaluating $f^{(i)}_{\mathbf{r}_i}$ at $3 \cdot 2^{v-j}$ points. It is not hard to see that $\mathcal{P}$ can evaluate $f^{(i)}_{\mathbf{r}_i}$ at any point in $O(S_i + S_{i+1})$ time using techniques similar to Lemma 1.8 from Lecture 4. This yields a runtime for $\mathcal{P}$ of $O(2^v \cdot (S_i + S_{i+1}))$. Over all $d$ layers of the circuit, $\mathcal{P}$'s runtime is bounded by $O(S^3)$.

***Method 2***: Cormode et al. [CMT12] improved on the $O(S^3)$ runtime of Method 1 by observing, just as in the matrix multiplication protocol from the previous lecture, that the $3 \cdot 2^{v-j}$ points at which $\mathcal{P}$ must evaluate $f^{(i)}_{\mathbf{r}_i}$ in round $j$ of the sum-check protocol are highly structured, in the sense that their trailing entries are Boolean. That is, it suffices for $\mathcal{P}$ to evaluate $f^{(i)}_{\mathbf{r}_i}(\mathbf{z})$ for all points $\mathbf{z}$ of the form: $\mathbf{z} = (r_1, \ldots, r_{j-1}, \{0,1,2\}, b_{j+1}, \ldots, b_v)$, where $v = 2s_{i+1}$ and each $b_k \in \{0,1\}$.

For each such point $\mathbf{z}$, the bottleneck in evaluating $f^{(i)}_{\mathbf{r}_i}(\mathbf{z})$ is in evaluating $\widetilde{\text{add}}_i(\mathbf{z})$ and $\widetilde{\text{mult}}_i(\mathbf{z})$. A direct application of Lemma 1.8 from Lecture 4 implies that each such evaluation can be performed in $2^v = O(S^2_{i+1})$ time. However, we can do much better by observing that the functions $\text{add}_i$ and $\text{mult}_i$ are *sparse*, in the sense that $\text{add}_i(\mathbf{a},\mathbf{b},\mathbf{c}) = \text{mult}_i(\mathbf{a},\mathbf{b},\mathbf{c}) = 0$ for all Boolean vectors $(\mathbf{a},\mathbf{b},\mathbf{c}) \in \mathbb{F}^v$ except for the $S_i$ vectors of the form $(\mathbf{a}, \text{in}_{1,i}(\mathbf{a}), \text{in}_{2,i}(\mathbf{a}))$: $\mathbf{a} \in \{0,1\}^{s_i}$.

Thus, we can write $\widetilde{\text{add}}_i(\mathbf{z}) = \sum_{\mathbf{a} \in \{0,1\}^{s_i}} \chi_{(\mathbf{a}, \text{in}_{1,i}(\mathbf{a}), \text{in}_{2,i}(\mathbf{a}))}(\mathbf{z})$, where the sum is only over addition gates $\mathbf{a}$ at layer $i$ of $\mathcal{C}$, and similarly for $\widetilde{\text{mult}}_i(\mathbf{z})$. Just as in the analysis of Method 2 for implementing the prover in the matrix multiplication protocol of the previous lecture, for any input $\mathbf{z}$ of the form $\mathbf{z} = (r_1, \ldots, r_{j-1}, \{0,1,2\}, b_{j+1}, \ldots, b_v)$, it holds that $\chi_{(\mathbf{a}, \text{in}_{1,i}(\mathbf{a}), \text{in}_{2,i}(\mathbf{a}))}(\mathbf{z}) = 0$ unless the last $v - j$ entries of $\mathbf{z}$ and $(\mathbf{a}, \text{in}_{1,i}(\mathbf{a}), \text{in}_{2,i}(\mathbf{a}))$ are equal (here, we are exploiting the fact that the trailing entries of $\mathbf{z}$ are Boolean). Hence, $\mathcal{P}$ can evaluate $\widetilde{\text{add}}_i(\mathbf{z})$ at all the necessary points $\mathbf{z}$ in each round of the sum-check protocol with a single pass over the gates at layer $i$ of $\mathcal{C}$: for each gate $\mathbf{a}$ in layer $i$, $\mathcal{P}$ only needs to update $\widetilde{\text{add}}_i(\mathbf{z}) \leftarrow \widetilde{\text{add}}_i(\mathbf{z}) + \chi_{(\mathbf{a}, \text{in}_{1,i}(\mathbf{a}), \text{in}_{2,i}(\mathbf{a}))}(\mathbf{z})$ for the three values of $\mathbf{z}$ whose trailing $v - j$ entries equal the trailing entries of $(\mathbf{a}, \text{in}_{1,i}(\mathbf{a}), \text{in}_{2,i}(\mathbf{a}))$.

## 4.2 Evaluating $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ Efficiently

The issue of the verifier efficiently evaluating $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ at a random points $\omega \in \mathbb{F}^{s_i + 2s_{i+1}}$ is a tricky one. While there does not seem to be a clean characterization of precisely which circuits have $\widetilde{\text{add}}_i$'s and $\widetilde{\text{mult}}_i$'s that can be evaluated in $O(\log S)$ time, most circuits that exhibit any kind of repeated structure satisfy this property. In particular, the papers [CMT12, Tha13] show that the evaluation can be computed in $O(s_i + s_{i+1}) = O(\log S)$ time for a variety of common wiring patterns and specific circuits. This includes the canonical circuit for simulating a space-bounded machine whose construction was sketched in Section 2. It also includes where the wiring patterns involve basic arithmetic on gate indices, and specific circuits computing functions such as MATMULT, pattern matching, Fast Fourier Transforms, and various problems of interest in the streaming literature, like frequency moments and distinct elements. Moreover, we will see next lecture that $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ can be evaluated efficiently for any circuit that operates in a *data parallel* manner (and the prover's runtime can even be reduced to $O(S)$ for such circuits).

In addition, various suggestions have been put forth for what to do when $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ cannot be evaluated in time $O(\log S)$. For example, as observed by Cormode et al. [CMT12], these computations can always be done by $\mathcal{V}$ in $O(\log S)$ *space* as long as the circuit is log-space uniform[1] which is sufficient

[1] Log-space uniform roughly means that the circuit $\mathcal{C}$ has a very succinct implicit representation. Specifically, for any layer $i$

in streaming applications where the space usage of the verifier is paramount [CMT12]. Moreover, these computations can be done offline before the input is even observed, because they only depend on the wiring of the circuit, and not on the input [GKR08, CMT12]. [Tha13] observes that if $\mathcal{C}$ is *data parallel*, meaning that it consists of many independent executions of the same subcomputation (possibly with aggregation of the results), then these computations take time at most proportional to the size of the sub-computation, and in particular grows at most logarithmically with the number of subcomputations.

Finally, in [GKR08] Goldwasser, Kalai, and Rothblum considered the option of outsourcing the computation of $\widetilde{\text{add}}_i(\mathbf{r}_i, \mathbf{b}^*, \mathbf{c}^*)$ and $\widetilde{\text{mult}}_i(\mathbf{r}_i, \mathbf{b}^*, \mathbf{c}^*)$ themselves. In fact, this option plays a central role in obtaining their result for general log-space uniform circuits. Specifically, GKR's result for general log-space uniform circuits are obtained via a two-stage proof. First, they give a protocol for any problem computable in (non-deterministic) logspace by applying their protocol to the canonical circuit for simulating a space-bounded Turing machine. This circuit has a highly regular wiring pattern for which $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ can be evaluated in $O(\log S)$ time.[2]

For a general log-space uniform circuit $\mathcal{C}$, it is not known how to identify low-degree extensions of $\text{add}_i$ and $\text{mult}_i$ that can be evaluated at $\omega$ in polylogarithmic time. Rather, Goldwasser et al. outsource computation of $\widetilde{\text{add}}_i(\mathbf{r}_i, \mathbf{b}^*, \mathbf{c}^*)$ and $\widetilde{\text{mult}}_i(\mathbf{r}_i, \mathbf{b}^*, \mathbf{c}^*)$ themselves. Since $\mathcal{C}$ is log-space uniform, $\widetilde{\text{add}}_i(\mathbf{r}_i, \mathbf{b}^*, \mathbf{c}^*)$ and $\widetilde{\text{mult}}_i(\mathbf{r}_i, \mathbf{b}^*, \mathbf{c}^*)$ can be computed in logarithmic space, and the protocol for logspace computations applies directly.

# 5 Leveraging Data Parallelism for Further Speedups

Data parallel computation refers to any setting in which the same sub-computation is applied independently to many pieces of data, before possibly aggregating the results. The protocol of this section makes no assumptions on the sub-computation that is being applied (in particular, it handles sub-computations computed by circuits with highly irregular wiring patterns), but does assume that the sub-computation is applied independently to many pieces of data. Figure 5 gives a schematic of a data parallel computation.

Data parallel computation is pervasive in real-world computing. For example, consider any *counting query* on a database. In a counting query, one applies some function independently to each row of the database and sums the results. For example, one may ask "How many people in the database satisfy Property $P$?" The protocol below allows one to verifiably outsource such a counting query with overhead that depends minimally on the size of the database, but that necessarily depends on the complexity of the property $P$. In later lectures, we will see that data parallel computations are in some sense "universal", in that efficient transformations from high-level computer programs to circuits often yield data parallel circuits.

**The Protocol and its Costs.** Let $C$ be a circuit of size $S$ with an arbitrary wiring pattern, and let $C^*$ be a "super-circuit" that applies $C$ independently to $B = 2^b$ different inputs before aggregating the results in some fashion. For example, in the case of a counting query, the aggregation phase simply sums the results of the

and gate label $\mathbf{a} \in \{0,1\}^{s_i}$, there is a logarithmic-space algorithm that is capable of determining all relevant information about gate $\mathbf{a}$ at layer $i$ of $\mathcal{C}$. That is, the algorithm can output the labels of all of $\mathbf{a}$'s neighbors, and is capable of determining if $\mathbf{a}$ is an addition gate or a multiplication gate.

[2] In [GKR08], Goldwasser et al. actually use higher degree extensions of $\text{add}_i$ and $\text{mult}_i$ obtained by arithemetizing a Boolean formula of size polylog$(S)$ computing these functions. The use of these extensions results in a prover whose runtime is a large polynomial in $S$ (i.e., $O(S^4)$). Cormode et al. [CMT12] observe that in fact the multilinear extensions of $\text{add}_i$ and $\text{mult}_i$ can be used for this circuit, and that with these extensions the prover's runtime can be brought down to $O(S \log S)$.

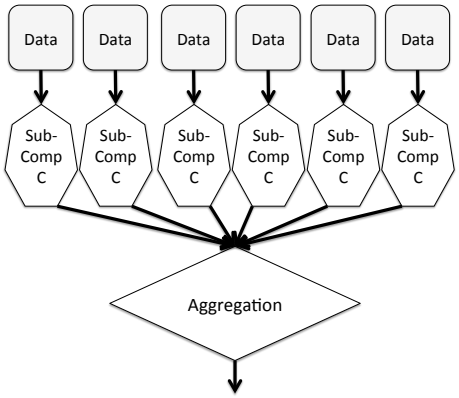


Figure 5: Schematic of a data parallel computation.

data parallel phase. Assume that the aggregation step is sufficiently simple that the aggregation itself can be verified using the techniques of Section 4.1.

If one naively applies the GKR protocol to the super-circuit $C^*$, $\mathcal{V}$ might have to perform an expensive pre-processing phase to evaluate the wiring predicates $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ of $C^*$ at the necessary locations—this would require time $\Omega(B \cdot S)$. Moreover, when applying the basic GKR protocol to $C^*$ using the techniques of [CMT12], $\mathcal{P}$ would require time $\Theta(B \cdot S \cdot \log(B \cdot S))$. A different approach was taken by Vu et al. [VSBW13], who applied the GKR protocol $B$ independent times, once for each copy of $C$. This causes both the communication cost and $\mathcal{V}$'s online check time to grow linearly with $B$, the number of sub-computations, which is undesirable.

In contrast, the protocol of this section (due to [WJB+17], building on [Tha13]) achieves the best of both both worlds, in that the overheads for the prover and verifier have no dependence on the number of inputs $B$ to which $C$ is applied. More specifically, the preprocessing time of the verifier is at most $O(S)$, independent of $B$. The prover runs in time $O(BS + S\log S)$. Observe that as long as $B > \log S$ (i.e., there is a sufficient amount of data parallelism in the computation), $O(BS + S\log S) = O(B \cdot S)$, and hence the prover is only a constant factor slower than the time required to evaluate the circuit gate-by-gate with no guarantee of correctness.

The idea of the protocol is that although each sub-computation $C$ can have a complicated wiring pattern, the circuit is maximally regular between sub-computations, as the sub-computations do not interact at all. It is possible to leverage this regularity to minimize the pre-processing time of the verifier, and to significantly speed up the prover.

## 5.1 Protocol Details

Let $\mathcal{C}$ be an arithmetic circuit over $\mathbb{F}$ of depth $d$ and size $S$ with an arbitrary wiring pattern, and let $\mathcal{C}^*$ be the circuit of depth $d$ and size $B \cdot S$ obtained by laying $B$ copies of $C$ side-by-side, where $B = 2^b$ is a power of 2. We will use the same notation as in Section 4, using $^*$'s to denote quantities referring to $\mathcal{C}^*$. For example, layer $i$ of $\mathcal{C}$ has size $S_i = 2^{s_i}$ and gate values specified by the function $W_i$, while layer $i$ of $\mathcal{C}^*$ has size $S_i^* = 2^{s_i^*} = 2^{b+s_i}$ and gate values specified by $W_i^*$.

Consider layer $i$ of $C^*$. Let $\mathbf{a} = (\mathbf{a}_1, \mathbf{a}_2) \in \{0,1\}^{s_i} \times \{0,1\}^b$ be the label of a gate at layer $i$ of $\mathcal{C}^*$, where $\mathbf{a}_2$ specifies which "copy" of $C$ the gate is in, while $\mathbf{a}_1$ designates the label of the gate within the copy. Similarly, let $\mathbf{b} = (\mathbf{b}_1, \mathbf{b}_2) \in \{0,1\}^{s_{i+1}} \times \{0,1\}^b$ and $\mathbf{c} = (\mathbf{c}_1, \mathbf{c}_2) \in \{0,1\}^{s_{i+1}} \times \{0,1\}^b$ be the labels of two

gates at layer $i+1$. The key to achieving the speedups for data parallel circuits relative to the interactive proof described in Section 4 is to tweak the expression in Proposition 4.1 for $\tilde{W}_i$. Specifically, Proposition 4.1 represents $\tilde{W^*}_i(\mathbf{z})$ as a sum over $\left(S^*_{i+1}\right)^2$ terms. In this section, we leverage the data parallel structure of $C^*$ to represent $\tilde{W^*}_i(\mathbf{z})$ as a sum over $S^*_{i+1} \cdot S_{i+1}$ terms, which is smaller than $\left(S^*_{i+1}\right)^2$ by a factor of $B$.

**Lemma 5.1.** *Let $h$ denote the polynomial $\mathbb{F}^{s_i \times k} \to \mathbb{F}$ defined via $h(\mathbf{a}_1, \mathbf{a}_2) := \sum_{\mathbf{b}_1, \mathbf{c}_1 \in \{0,1\}^{s_{i+1}}} g(\mathbf{a}_1, \mathbf{a}_2, \mathbf{b}_1, \mathbf{c}_1)$, where*

$$g(\mathbf{a}_1, \mathbf{a}_2, \mathbf{b}_1, \mathbf{c}_1) := \widetilde{\text{add}}_i(\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1) \left( \widetilde{W}^*_{i+1}(\mathbf{b}_1, \mathbf{a}_2) + \widetilde{W}^*_{i+1}(\mathbf{c}_1, \mathbf{a}_2) \right) + \widetilde{\text{mult}}_i(\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1) \cdot \widetilde{W}^*_{i+1}(\mathbf{b}_1, \mathbf{a}_2) \cdot \widetilde{W}^*_{i+1}(\mathbf{c}_1, \mathbf{a}_2).$$

*Then $h$ extends $W^*_i$.*

Essentially, Lemma 5.1 says that an addition (respectively, multiplication) gate $\mathbf{a} = (\mathbf{a}_1, \mathbf{a}_2) \in \{0,1\}^{s_i + b}$ is connected to gates $\mathbf{b} = (\mathbf{b}_1, \mathbf{b}_2) \in \{0,1\}^{s_{i+1}+b}$ and $\mathbf{c} = (\mathbf{c}_1, \mathbf{c}_2) \in \{0,1\}^{s_{i+1}+b}$ if and only if $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ are all in the same copy of $\mathcal{C}$, and $\mathbf{a}$ is connected to $\mathbf{b}$ and $\mathbf{c}$ within the copy.

**Lemma 5.2.** *(Restatement of [Rot09, Lemma 3.2.1].) For* any *polynomial $h\colon \mathbb{F}^{s_i} \to \mathbb{F}$ extending $W_i$, the following polynomial identity holds:*

$$\tilde{W}_i(\mathbf{z}) = \sum_{\mathbf{a} \in \{0,1\}^{s_i}} \widetilde{\beta}_{s_i}(\mathbf{z}, \mathbf{a}) h(\mathbf{a}). \tag{3}$$

*Proof.* It is easy to check that the right hand side of Equation (3) is a multilinear polynomial in $\mathbf{z}$, and that it agrees with $W_i$ on all Boolean inputs. Thus, the right hand side of Equation (3), viewed as a polynomial in $\mathbf{z}$, must be the multilinear extension $\tilde{W}_i$ of $W_i$. □

Combining Lemmas 5.1 and 5.2 implies that for any $\mathbf{z} \in \mathbb{F}^{s^*_i}$,

$$\tilde{W}^*_i(\mathbf{z}) = \sum_{(\mathbf{a}_1, \mathbf{a}_2, \mathbf{b}_1, \mathbf{c}_1) \in \{0,1\}^{s_i + b + 2s_{i+1}}} g^{(i)}_{\mathbf{z}}(\mathbf{a}_1, \mathbf{a}_2, \mathbf{b}_1, \mathbf{c}_1), \tag{4}$$

where

$$g^{(i)}_{\mathbf{z}}(\mathbf{a}_1, \mathbf{a}_2, \mathbf{b}_1, \mathbf{c}_1) :=$$

$$\widetilde{\beta}(\mathbf{z}, (\mathbf{a}_1, \mathbf{a}_2)) \cdot \left[ \widetilde{\text{add}}_i(\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1) \left( \widetilde{W}^*_{i+1}(\mathbf{b}_1, \mathbf{a}_2) + \widetilde{W}^*_{i+1}(\mathbf{c}_1, \mathbf{a}_2) \right) + \widetilde{\text{mult}}_i(\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1) \cdot \widetilde{W}^*_{i+1}(\mathbf{b}_1, \mathbf{a}_2) \cdot \widetilde{W}^*_{i+1}(\mathbf{c}_1, \mathbf{a}_2) \right].$$

Thus, to reduce a claim about $\tilde{W}^*_i(\mathbf{r}_i)$ to a claim about $\tilde{W}^*_{i+1}(\mathbf{r}_{i+1})$, it suffices to apply the sum-check protocol to the polynomial $g^{(i)}_{\mathbf{r}_i}$.

**Costs for $\mathcal{V}$.** To bound $\mathcal{V}$'s runtime, observe that $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ can be evaluated at a random point in $\mathbb{F}^{s_i + 2s_{i+1}}$ in pre-processing in time $O(S_i)$ by enumerating the in-neighbors of each of the $S_i$ gates at layer $i$ in order to apply Lemma 1.8 from Lecture 4. Adding up the pre-processing time across all iterations $i$ of our protocol, $\mathcal{V}$'s pre-processing time is $O(\sum_i S_i) = O(S)$ as claimed.

**Costs for $\mathcal{P}$.** The insights that go into implementing the honest prover in time $O(B \cdot S + S \log S)$ build on ideas related the Method 3 for implementing the prover in the Matrix Multiplication protocol of Lecture 7, and heavily exploit the fact that Equation (4) represents $\tilde{W^*}_i(\mathbf{z})$ as a sum over just $S^*_{i+1} \cdot S_{i+1}$ terms, rather than the $\left(S^*_{i+1}\right)^2$ terms appearing in Equation (4.1).

| Communication | Rounds | $\mathcal{V}$ time | $\mathcal{P}$ time |
|---|---|---|---|
| $O(d \cdot \log(B \cdot S))$ field elements | $O(d \cdot (\log(B \cdot S)))$ | online time: $O(B \cdot n + d \cdot (\log(B \cdot S)))$ pre-processing time: $O(S)$ | $O(B \cdot S + S \cdot \log(S))$ |

Table 2: Costs of the IP of Section 5 when applied to any log-space uniform arithmetic circuit $\mathcal{C}$ of size $S$ and depth $d$ over $n$ variables, that is applied $B$ times in a data parallel manner (cf. Figure 5).

# References


[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In Shafi Goldwasser, editor, *ITCS*, pages 90–112. ACM, 2012.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, STOC '08, pages 113–122, New York, NY, USA, 2008. ACM.

[LFKN92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39:859–868, October 1992.

[Rot09] Guy Rothblum. *Delegating computation reliably : paradigms and constructions*. PhD thesis, Massachusetts Institute of Technology, 2009.

[Sha92] Adi Shamir. IP = PSPACE. *J. ACM*, 39:869–877, October 1992.

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Proceedings of the 33rd Annual Conference on Advances in Cryptology*, CRYPTO'13, Berlin, Heidelberg, 2013. Springer-Verlag.

[Tha15] Justin Thaler. A note on the gkr protocol, 2015. Available at: `http://people.cs.georgetown.edu/jthaler/GKRNote.pdf`.

[VSBW13] Victor Vu, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 223–237. IEEE Computer Society, 2013.

[WJB+17] Riad S Wahby, Ye Ji, Andrew J Blumberg, Abhi Shelat, Justin Thaler, Michael Walfish, and Thomas Wies. Full accounting for verifiable outsourcing. *IACR Cryptology ePrint Archive*, 2017:242, 2017.