Brief Announcement: Incrementally Verifiable Distributed Computation

Eden Aldema Tshuva
Tel Aviv University, Israel

Rotem Oshman
Tel Aviv University, Israel

Abstract

Incrementally verifiable computation (IVC) is a cryptographic scheme that allows a prover to certify the correctness of a long or ongoing computation in an incremental manner, by repeatedly updating a proof certifying the computation so far. Updating the proof does not require access to the entire trace of the computation, which makes the IVC prover memory efficient.

In this work we construct incrementally verifiable *distributed* computation, which allows a distributed algorithm to efficiently certify its own execution using low memory and communication overhead. Our primary motivation is massively-parallel computation (MPC), where memory efficiency is make-or-break: the machines participating in an MPC algorithm usually cannot store the entire trace of their computation. Thus, certifying MPC algorithms essentially *requires* distributed IVC.

At the heart of this work is a new abstraction, *updatable batch arguments for* NP (UpBARGs), which we define and construct. Standard BARGs allow one to prove a batch of k NP-statements using a proof whose length barely grows with k; however, the statements and their witnesses must all be known in advance. In contrast, UpBARGs support adding statements and witnesses on the fly, making them a flexible tool for constructing IVC across different computational models. We use UpBARGs to construct IVC for streaming algorithms, for MPC algorithms, and for PRAM algorithms in the exclusive-read exclusive-write (EREW) model.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols

Keywords and phrases Incrementally verifiable computation, massively parallel computation, streaming, parallel RAM, batch arguments, SNARG

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.44

Funding *Eden Aldema Tshuva*: Research supported by the Israeli Science Foundation, Grant No. 2338/23, and by AFOSR award FA9550-24-1-0156.
Rotem Oshman: Research funded by the Israel Science Foundation, Grant No. 2801/20, and by AFOSR award FA9550-24-1-0156.

1 Introduction

Incrementally verifiable computation (IVC), introduced by Valiant [15], enables a prover to generate a succinct proof that a long computation has been executed correctly, and update the proof incrementally as the computation progresses: given a proof that the first t steps have been executed correctly (for some t), the prover can update it to obtain a proof that the first $t+1$ steps have been executed correctly, without having to store in memory the entire trace of the computation so far. This makes IVC a natural fit for scenarios where long computations must be executed reliably over time: for example, resuming a paused computation from an intermediate verified state, or continuously auditing the correctness of outsourced computation performed by an untrusted server or cloud.

The incremental structure of IVC is particularly well-suited to computations that are *distributed* or *reactive* in nature: where inputs arrive over time, or are jointly held by multiple parties that collaborate to carry out the computation. In this work we extend the scope of

IVC to encompass such distributed and reactive models of computation, bringing the benefits of incremental verification to a setting that arguably stands to gain from it the most. We construct IVC for massively parallel computation (MPC), streaming algorithms and PRAM. Our main technical tool is *updatable hash-and-*BARG *for* NP(UpBARGs), a generic building block that we believe is of independent interest: it allows us to construct IVC for models where we can prove the correctness of a *single* computation step.

IVC for reactive and distributed computation. An IVC scheme consists of a *prover* and a *verifier*. The prover is intended to run alongside the computation whose correctness it proves, so it is important for the prover to have low overhead in terms of space, and for distributed computations, also in terms of communication and rounds. At each point in time, the prover holds a proof that the computation has executed correctly *so far*, and as the computation proceeds, the prover updates the proof. The interface to the prover consists of a *proof update* procedure that takes a proof π_t reflecting some prefix of t computation steps, alongside the current state of the computation st_t, and returns a new proof π_{t+1} reflecting $t+1$ computation steps. The verifier is modeled as a sequential algorithm. Because the input to the system could be very large (e.g., in streaming or MPC algorithms), we do not assume that the verifier can store all of it, or even that the verifier can store a single global state of the system in its entirety. Instead, the verifier is presented with *hashes* h_0, h_t of two states $\mathsf{st}_0, \mathsf{st}_t$ of the computation (respectively), a number of steps t, and a proof π, and it checks whether π is a "convincing" proof that the computation indeed reaches st_t after t computation steps from st_0. We use the short-hand notation "$\mathsf{st}_0 \xrightarrow{t} \mathsf{st}_t$" for this statement. The notions of "states" and "computation steps" depends on the exact computation model (e.g., it can mean synchronous rounds, or a single step of a process).

The fact that the verifier is given *hashes* h_0, h_t of the states $\mathsf{st}_0, \mathsf{st}_t$ instead of plaintext, presents some definitional subtleties, as one cannot reconstruct the original state from its hash value; technically, given h_0, h_t and t, the verifier does not even *know* what statement "$\mathsf{st}_0 \xrightarrow{t} \mathsf{st}_t$" is being asserted. To resolve this we adopt definitions analogous to the one used for RAM delegation [11, 6], where the same issue arises. Informally, the distributed IVC schemes that we design have the following essential properties:

- Succinctness: the proof is short enough to store on a single machine.
- Incremental completeness: for any system states $\mathsf{st}_0, \mathsf{st}_t$ and st_{t+1} such that st_t transitions to st_{t+1} in one computation step, if the prover is given $\mathsf{st}_0, \mathsf{st}_t$ and a proof π_t such that the verifier accepts π_t as a proof for the statement $\mathsf{st}_0 \xrightarrow{t} \mathsf{st}_t$, then the prover produces a proof π_{t+1} for the statement "$\mathsf{st}_0 \xrightarrow{t+1} \mathsf{st}_{t+1}$", which is also accepted by the verifier.[1]
- Computational soundness: no poly-size algorithm[2] can generate values h_0, h_t, h'_t, a step number t and two proofs π_t, π'_t, such that $h_t \neq h'_t$, but the verifier accepts both the proof π_t with h_0, h_t, t and the proof π'_t with h_0, h'_t, t. Intuitively, this means that the verifier cannot be convinced of two "contradictory statements", asserting that starting from some initial state (reflected by the hash h_0), after t computation steps the system ends up in two different states (reflected by the hashes $h_t \neq h'_t$).

The main and most challenging model we handle in this work is *massively parallel computation* (MPC) [7, 12], where n space-bounded machines collaborate to compute over an input initially partitioned between them. Proving the correctness of MPC computations

[1] Here, by "the verifier accepts π_t for $\mathsf{st}_0 \xrightarrow{t} \mathsf{st}_t$" we mean it accepts (h_0, h_t, t, π) where h_0, h_t are the respective hash values of $\mathsf{st}_0, \mathsf{st}_t$.

[2] That is, an algorithm that can be represented as a circuit with a polynomial number of wires. This is at least as strong as probabilistic polynomial-time Turing machines.

almost *requires* IVC: non-incremental verification typically requires access to the entire trace of the computation at once, which in the MPC model no single machine can store. Thus, while efficient distributed provers have already been developed for other distributed models [1, 2], no such construction is known for MPC, even if we do not insist on IVC and are willing to settle for a non-incremental construction. Thus, our construction of IVC for MPC addresses not only the problem of incrementality but also the more basic problem of proving the correctness of massively parallel computation.

We consider two use cases. (1) Reliable cloud delegation: a user with input x outsources a heavy computation to a server farm or cloud provider, which returns both the output and a short correctness certificate. The user (a single weak machine) hashes x once, never rereads it, and verifies the result in polylogarithmic time in $|x|$. (2) Internal MPC verification: the network itself verifies correctness so far (e.g., to detect faulty machines). Our distributed IVC maintains a short, continuously updatable certificate that any machine can use to verify locally against its own input. This resembles proof labeling schemes [13] but with polylogarithmic, rather than overall linear proof growth, and with each machine verifying independently without communication.

Our framework for constructing IVC is modular and generic: we are able to construct IVC for practically any model of deterministic computation for which we can prove the correctness of a single computation step. We demonstrate this by constructing, in addition to the IVC for MPC discussed above, IVC for streaming algorithms, where a low-space single machine processes a stream of elements that arrive over time and cannot all be stored at once, and for PRAM algorithms in the exclusive-read exclusive-write model (EREW), which captures parallel algorithms in shared memory. While for streaming algorithms, proving the correctness of a single computation step is trivial (as the step itself is represented succinctly by the state transition and the stream element), for the PRAM and MPC models we use cryptographic proof systems called *succinct non-interactive arguments* (SNARGs): essentially, these are the non-incremental analog of IVC. Specifically, for PRAM we use the existing SNARG for PRAM of [9]. For MPC, no such construction is known, so we first construct a SNARG for one-round MPC, and then lift it into full-fledged IVC for MPC. Unlike some commonly used heuristically sound SNARKs (SNARGs of knowledge) and IVC [3, 10], all of our constructions are sound under *standard* cryptographic assumptions.

Our main building block: updatable hash-and-BARG for NP (UpBARGs). Proving that a computation $\mathsf{st}_0 \xrightarrow{t} \mathsf{st}_t$ is correct boils down to proving the conjunction of t statements: "for each $i = 0, \ldots, t-1$, the system transitions from state st_i to state st_{i+1} in one step". In existing constructions of (non-incremental) SNARG from standard cryptographic assumptions [6, 16, 4], this is done using two cryptographic tools: *batch arguments (*BARG*s) for* NP and *hash families with local openings* (e.g., Merkle trees). Informally, a BARG for an NP-language $\mathcal{L}$ allows us to prove the conjunction of k statements, each of the form "$x_i \in \mathcal{L}$", using a proof whose length grows linearly with the length of a *single* NP-witness, and only polylogarithmically with the number of statements k. A hash family with local openings allows us to hash an n-sized vector v to a short hash value h, and in addition, given an index i, to produce a *short opening* ρ_i to the i-th index. Given only (h, i, v_i, ρ), one can verify that x_i is indeed the i-th entry of the vector whose hash is h.

In prior constructions, to prove (non-incrementally) that $\mathsf{st}_0 \xrightarrow{t} \mathsf{st}_t$, the prover hashes the vector $(\mathsf{st}_0, \ldots, \mathsf{st}_t)$ using a hash with local openings to obtain a hash value h. Then it constructs a BARG asserting that for each $i = 0, \ldots, t-1$, there exist states $\mathsf{st}_i, \mathsf{st}_{i+1}$ and openings ρ_i, ρ_{i+1} such that h opens to $\mathsf{st}_i, \mathsf{st}_{i+1}$ in positions $i, i+1$ (respectively), and st_i transitions to st_{i+1} in one step. Here, the states $\mathsf{st}_i, \mathsf{st}_{i+1}$ and the openings ρ_i, ρ_{i+1} serve as an NP-witness for the statement "the i-th step is reflected correctly in the hash h".

When incrementally proving the correctness of a reactive or distributed computation, we do not know in advance *what* statements the prover will need to prove, as the state that the system will reach depends on the future inputs or messages. Thus, we cannot lay out the entire computation trace in advance, hash it, and prepare a BARG proof. To overcome this, we design a scheme we call *updatable hash-and-*BARG (UpBARG), which imitates the above hash-and-BARG paradigm while allowing both the hash and the BARG to be constructed incrementally, together. This scheme enables us to update a hash value and a proof for a conjunction of NP-statements on-the-fly, without advance knowledge. To use our UpBARG to construct IVC, we must overcome another difficulty, which is that without having the entire trace in advance, we cannot compute the openings that serve as NP-witnesses even for the entries we already have. We show that we can compute the required openings *in hindsight*, using properties of tree-based hash families with local openings. Our techniques are inspired by the recent works on IVC for sequential deterministic computation [14, 8].

2 Brief Preliminaries

The common reference string model and computational hardness. Our work is set in the *common reference string* (CRS) model. In this model, all parties have access to a string that is sampled randomly by a trusted setup process, denoted by Gen, which takes a *security parameter* λ. (This can be viewed as public randomness.) The security parameter governs the computational resources that must be invested to break the security of the protocol: we say that a task that involves the CRS is *computationally hard* if given $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda)$, no poly-size (in λ) adversary can succeed in the task, except with *negligible probability* in λ.

Batch arguments for NP (BARGs). A batch argument (BARG) [5] allows a prover to succinctly prove the conjunction of k NP statements, where the size of the proof barely grows with k, and is similar to the size of one witness. The interface is as follows:

- $\mathcal{P}(\mathsf{crs}, \mathcal{M}, x_1, \ldots, x_k, w_1, \ldots, w_k) \to (\pi)$: takes a reference string crs, a machine $\mathcal{M}$, instances $x_1, \ldots, x_k$, and witnesses $w_1, \ldots, w_k$, and outputs a proof π.
- $\mathcal{V}(\mathsf{crs}, \mathcal{M}, \pi) \to b$: takes a reference string crs, a machine $\mathcal{M}$, and a proof π and outputs an acceptance bit b.

We say a BARG is *somewhere extractable* if it allows the extraction of *one witness* from a convincing proof π, as follows:

- The crs generation procedure Gen may be called in *trapdoor mode*. In trapdoor mode, Gen takes as additional input an index $i \in [k]$, called the *binding index*. It outputs a pair $(\mathsf{crs}, \mathsf{td}_i)$, where td is a *trapdoor* that can later be used to recover the i-th witness. In trapdoor mode, the Gen procedure has a property called *index hiding*: given crs (without td_i), it is computationally hard to *find* the binding index i.
- The BARG has the *somewhere argument of knowledge* property: There exists an efficient auxiliary *extraction* procedure, $\mathcal{E}(\mathsf{td}_i, \mathcal{M}, \pi) \to w_i$, which satisfies the following. Suppose we call Gen in trapdoor mode with a binding index i, and obtain $(\mathsf{crs}, \mathsf{td}_i)$. Given only crs, it is computationally hard to find a proof π that is accepted by the verifier, such that when we extract a witness w_i using $\mathcal{E}(\mathsf{td}_i, \mathcal{M}, \pi)$, we have $\mathcal{M}(x_i, w) \neq 1$.

3 Technical Overview

We give an overview of our distributed IVC constructions, with emphasis on the MPC model. Our framework for constructing IVC is generic and quite simple once we have (1) a SNARG for one-step computations in the model, and (2) an updatable hash-and-BARG scheme.

3.1 Defining and Using Updatable Hash-and-BARG

An updatable hash-and-BARG (UpBARG) is defined with respect to some hash family with local openings. It consists of three algorithms, $(\mathsf{Gen}, \mathcal{U}, \mathcal{V})$, where Gen is a standard CRS generation algorithm, and $\mathcal{U}, \mathcal{V}$ are update and verification algorithms, respectively. The syntax for the update and verification algorithms is as follows:

- $\mathcal{U}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, x, w) \to (H^*, \Pi^*)$: takes as input a hash key hk, a common reference string crs, a Turing machine $\mathcal{M}$, a hash value H, a proof Π, and a new statement x and witness w, and outputs a hash value H and a new proof Π^*.
- $\mathcal{V}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi) \to b$: takes a hash key hk, a common reference string crs, a Turing machine $\mathcal{M}$, a hash value H, and proof Π, and outputs an acceptance bit $b \in \{0, 1\}$.

Our UpBARG satisfies the standard succinctness and index hiding properties of BARGs (see Section 2), but it has stronger completeness and argument of knowledge properties:

- *Incremental completeness:* if $\mathcal{V}$ accepts $(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi)$, and $\mathcal{M}(x, w) = 1$, then it also accepts $(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H^*, \Pi^*)$, where $(H^*, \Pi^*) = \mathcal{U}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, x, w)$.
- *Somewhere argument of knowledge:* upon using the trapdoor version of Gen, with binding index i, the extractor $\mathcal{E}$ extracts not only an instance-witness pair (x, w), but also an opening ρ proving that H indeed opens to x in the index i.

Supporting consistency checks. Our IVC requires the ability to claim that two extracted instances are *consistent* with each other, for a definition of "consistent" that is supplied by the user. To that end, we extend the syntax above to feed the update and verification algorithms additional inputs: a Turing machine $\mathcal{C}$ specifying the consistency check, and a "last instance" x^-. Our incremental completeness guarantee in this case holds whenever $\mathcal{C}(x^-, x) = 1$. The extractor of the somewhere argument of knowledge guarantee now additionally extracts a *previous instance* x^- and a respective opening, such that: (1) H opens to x^- in location $i - 1$ and this is proved by said opening, and (2) $\mathcal{C}(x^-, x) = 1$.

From UpBARG to IVC. We now aim to incrementally certify the correctness of execution of an ongoing computation, denote it by $\mathcal{M}$, where we wish the proof updating procedure to be done in the same model as $\mathcal{M}$. Given a SNARG for $\mathcal{M}$ which prover's run in the same model, the UpBARG provides the required building block fairly directly. Let $\mathsf{Gen}, \mathcal{P}, \mathcal{V}$ be a SNARG scheme in which we are able to prove (within the computation model) the correctness of one step $\mathsf{st}_i \to \mathsf{st}_{i+1}$. Let $\mathcal{M}_{\mathcal{V}}$ be the non-deterministic machine which verifies the SNARG proof. That is, on inputs (x, w), $\mathcal{M}_{\mathcal{V}}$ interprets x as $(\mathsf{st}_i, \mathsf{st}_{i+1})$, and w as a SNARG-proof π for the transition $\mathsf{st}_i \to \mathsf{st}_{i+1}$, then it applies the SNARG verifier and accepts or rejects accordingly.

The IVC prover runs alongside $\mathcal{M}$ and updates an UpBARG proof Π for the machine $\mathcal{M}_{\mathcal{V}}$, and whenever the algorithm transitions from state st to state st', it uses $\mathcal{P}$ to prove $\mathsf{st} \to \mathsf{st}'$ and then updates Π with the instance-witness pair $((\mathsf{st}, \mathsf{st}'), \pi)$. We use consistency checks to ensure that every two consecutive statements overlap on their last and first states, respectively, to prevent a situation where we correctly assert that "$\mathsf{st}_1 \to \mathsf{st}_2$" and then "$\mathsf{st}'_2 \to \mathsf{st}_3$", but $\mathsf{st}_2 \neq \mathsf{st}'_2$, so we do not truly prove that $\mathsf{st}_1 \xrightarrow{2} \mathsf{st}_3$. Consistency checks allow us to force $\mathsf{st}_2 = \mathsf{st}'_2$, so that the conjunction of all statements together asserts the correctness of the entire multi-step computation from the initial state to the current one.

3.2 SNARG for One-Round Massively Parallel Computation

Our generic "SNARG-to-IVC via UpBARG" method can lift a SNARG for one computation step in a computational model into a full IVC for that model. However, for the MPC model, no SNARG (even for a single round) was known prior to our work. We outline our construction of such a SNARG, using UpBARGs again.

One MPC round as a RAM program. We observe that one round of MPC on n machines can be described as a RAM program, which takes a vector of the machines' initial states $(\mathsf{st}(1), \ldots, \mathsf{st}(n))$, and outputs the machines' states in the end of the round $((\mathsf{st}'(1), \ldots, \mathsf{st}'(n))$. Each state $\mathsf{st}(i)$ or $\mathsf{st}'(i)$ is an s-bit vector, where s is the space bound of an individual machine. To construct a SNARG for one-round-MPC, we imitate a SNARG for the program above. However, while a RAM program is executed on a single machine that can access any place in its random-access memory, in the MPC model, the "memory" is partitioned across the n machines. Fortunately, existing RAM SNARGs have the property that constructing the proof does not require access to the entire memory configuration: it is enough to have a *hash with local openings* of the memory, and openings to the locations that are accessed by the algorithm in each step. Thus, to construct our desired RAM-SNARG-like proof for the MPC model, we need to go through the following three stages: (1) Obtain a hash of the state vector $(\mathsf{st}(1), \ldots \mathsf{st}(n))$; (2) For each machine $i \in [n]$, obtain openings of the above hash value to the messages $m_1^i, \ldots, m_k^i$ (where $k \leq s$) that machine i receives during the round[3]; and (3) Using the openings, construct a SNARG proof which proves the transition $(\mathsf{st}(1), \ldots, \mathsf{st}(n)) \to (\mathsf{st}'(1), \ldots, \mathsf{st}'(n))$. The main challenge in this scheme is that each stage must be implemented in the MPC network: we do not *truly* have random access to the state assignments, because these are actual states of individual machines. For all three stages, our solution is based on having the n machines simulate some process on a tree-structured network. This network is an s-ary tree with n leaves which represent the states of the n actual machines, and the rest of the nodes are virtual nodes, simulated by the n real machines.

Using a virtual tree-structured network. To obtain a hash of the state assignment $(\mathsf{st}(1), \ldots, \mathsf{st}(n))$, we aggregate the hash up the tree: the leaves send their state to their parents, and each inner node, upon receiving values from its s children, hashes the values it received and sends them up to its parent. The root of the tree then obtains a hash of the entire state assignment. In order for each machine $i \in [n]$ to obtain an opening to its state $\mathsf{st}(i)$, we proceed this time down the tree, with each inner node receiving from its parent a partial opening down to its index, extending it to an opening for each of its children's indices, and sending each extended opening to the corresponding child. Each inner node is able to extend the opening because it is the one that hashed all of its children's values together. Eventually, each machine $i \in [n]$ receives from its parent a complete opening down to index i. Finally, we construct a SNARG proof for the alleged RAM computation $(\mathsf{st}(1), \ldots, \mathsf{st}(n)) \to (\mathsf{st}'(1), \ldots, \mathsf{st}'(n))$. To do so, we first observe that this transition is defined by n separate computations, and for each $i \in [n]$, we prove that when the network state assignment is $(\mathsf{st}(1), \ldots, \mathsf{st}(n))$, the next state of machine i is $\mathsf{st}'(i)$. Each such statement can be proved by a SNARG computed at machine i, as follows. Let j be a machine that sends message m_j^i to machine i during the round. Recall that after the previous stage, machine j

[3] We assume for simplicity that the messages each machine sends are stored as part of its state at the beginning of the round.

has an opening from the global hash to its state $\mathsf{st}(j)$. Machine j now computes an opening from $\mathsf{st}(j)$ to m_j^i and sends this opening to machine i.Each machine, upon receiving these openings, uses a RAM SNARG to prove that its transition $\mathsf{st}(i) \to \mathsf{st}(i+1)$ is legal.

The remaining challenge is to aggregate the individual SNARG proofs of the machines into one global succinct proof while maintaining soundness. To do this we use the virtual network again: we proceed up the tree, with each node sending its current proof up to its parent. Upon receiving the s proofs from its children, each inner node uses an UpBARG to obtain a proof of the *conjunction* of its childrens' statements. Finally, at the root of the tree, we obtain our one succinct proof for the entire transition $(\mathsf{st}(1), \ldots, \mathsf{st}(n)) \to (\mathsf{st}^+(1), \ldots, \mathsf{st}^+(n))$.

References

1 Eden Aldema Tshuva, Elette Boyle, Ran Cohen, Tal Moran, and Rotem Oshman. Locally verifiable distributed SNARGs. In *TCC*, pages 65–90. Springer, 2023. doi:10.1007/978-3-031-48615-9_3.

2 Eden Aldema Tshuva and Rotem Oshman. Fully Local Succinct Distributed Arguments. In *DISC*, pages 1:1–1:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.DISC.2024.1.

3 Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *STOC*, pages 111–120, 2013. doi:10.1145/2488608.2488623.

4 Arka Rai Choudhuri, Sanjam Garg, Abhishek Jain, Zhengzhong Jin, and Jiaheng Zhang. Correlation intractability and snargs from sub-exponential ddh. In *Crypto*, pages 635–668. Springer, 2023. doi:10.1007/978-3-031-38551-3_20.

5 Arka Rai Choudhuri, Abhishek Jain, and Zhengzhong Jin. Non-interactive batch arguments for NP from standard assumptions. In *Annual International Cryptology Conference*, pages 394–423. Springer, 2021. doi:10.1007/978-3-030-84259-8_14.

6 Arka Rai Choudhuri, Abhishek Jain, and Zhengzhong Jin. SNARGs for P from LWE. In *FOCS*, pages 68–79, 2021.

7 Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. doi:10.1145/1327452.1327492.

8 Lalita Devadas, Rishab Goyal, Yael Kalai, and Vinod Vaikuntanathan. Rate-1 non-interactive arguments for batch-NP and applications. In *FOCS*, pages 1057–1068, 2022. doi:10.1109/FOCS54457.2022.00103.

9 Cody Freitag, Rafael Pass, and Naomi Sirkin. Parallelizable delegation from LWE. In *TCC*, pages 623–652, 2022. doi:10.1007/978-3-031-22365-5_22.

10 Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016. doi:10.1007/978-3-662-49896-5_11.

11 Yael Kalai and Omer Paneth. Delegating ram computations. In *TCC*, pages 91–118. Springer, 2016.

12 Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948. SIAM, 2010. doi:10.1137/1.9781611973075.76.

13 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. In *PODC*, pages 9–18, 2005. doi:10.1145/1073814.1073817.

14 Omer Paneth and Rafael Pass. Incrementally verifiable computation via rate-1 batch arguments. In *FOCS*, pages 1045–1056, 2022. doi:10.1109/FOCS54457.2022.00102.

15 Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *TCC*, pages 1–18. Springer, 2008. doi:10.1007/978-3-540-78524-8_1.

16 Brent Waters and David J Wu. Batch arguments for NP and more from standard bilinear group assumptions. In *Crypto*, pages 433–463. Springer, 2022. doi:10.1007/978-3-031-15979-4_15.