

The Parallel Dynamic Complexity of the Abelian Cayley Group Membership Problem

V. Arvind   

The Institute of Mathematical Sciences (HBNI), Chennai, India
Chennai Mathematical Institute, India

Samir Datta   

Chennai Mathematical Institute and UMI ReLaX, India

Asif Khan  


Chennai Mathematical Institute, India

Shivdutt Sharma  

Indian Institute of Information Technology, Una, India

Yadu Vasudev  

Indian Institute of Technology Madras, Chennai, India

Shankar Ram Vasudevan 

Chennai Mathematical Institute, India

Abstract

Let G be a finite group given as input by its multiplication table. For a subset $S \subseteq G$ and an element $g \in G$ the *Cayley Group Membership Problem* (CGM) is to check if g belongs to the subgroup generated by S . While this problem is easily seen to be in polynomial time, pinpointing its parallel complexity has been of research interest over the years. Barrington et al [6] have shown that for abelian groups the CGM problem can be solved in $O(\log \log |G|)$ parallel time. In this paper we further explore the parallel complexity of the abelian CGM problem, with focus on the dynamic setting: the generating set S changes with insertions and deletions and the goal is to maintain a data structure that supports efficient membership queries to the subgroup $\langle S \rangle$. Though the static version of the CGM problem can be easily reduced to digraph reachability, the reduction does not carry over to the dynamic setting. We obtain the following results:

1. First, we consider the more general problem of Monoid Membership, where G is a monoid input by its multiplication table. When G is a *commutative monoid* we show there is a deterministic dynamic AC^0 algorithm¹ for membership testing that supports $O(1)$ insertions and deletions in each step.
2. Building on the previous result we show that there is a dynamic randomized AC^0 algorithm for abelian CGM that supports $\text{polylog}(|G|)$ insertions/deletions to S in each step.
3. If the number of insertions/deletions is at most $O(\log n / \log \log n)$ then we obtain a deterministic dynamic AC^0 algorithm for abelian CGM.
4. Applying these algorithms we obtain analogous results for the dynamic abelian Group Isomorphism.

We can also handle sub-linearly many changes to the multiplication table for G , utilizing the hamming distance between multiplication tables of any two distinct groups.

2012 ACM Subject Classification Theory of computation \rightarrow Parallel algorithms

Keywords and phrases Dynamic Complexity, Group Theory, Cayley Group Membership, Abelian Group Isomorphism

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2024.4

Related Version *Full Version*: <https://arxiv.org/abs/2308.10073>

Funding *Samir Datta*: Partially supported by a grant from Infosys foundation.

Asif Khan: Partially supported by a grant from Infosys foundation.

¹ Equivalently, a constant time parallel algorithm using polynomially many processors.



1 Introduction

The main algorithmic problem of interest in this paper, is the *Cayley Group Membership Problem* (CGM): Given as input a finite group G by its multiplication table (also known as its Cayley table), a subset $S \subseteq G$ and an element $g \in G$, test if $g \in \langle S \rangle$, where $\langle S \rangle$ is the subgroup of G generated by the elements in S .

The CGM problem was brought into focus by the work of Barrington et al [6] which raises intriguing questions about its parallel complexity.

Background. Membership testing in finite groups is well-studied [28]. Its computational complexity significantly depends on how G is given as input and on its elements' representation. For example, if the elements of G are represented as permutations on $[n] = \{1, 2, \dots, n\}$, then G is a subgroup of S_n , the group of all permutations on $[n]$. A natural compact description of G as input is by a generating set, as every finite group G has a generating set of size at most $\log |G|$. In this form, membership testing in *permutation groups* has been studied since the 1970's, pioneered by the work of Sims [29, 28]. There are efficient polynomial (in n) time algorithms for the problem as well as parallel algorithms for it. The problem is in NC: it can be solved in $\text{polylog}(n)$ time with polynomially many processors [4]. On the other hand, $G = \langle a \rangle$ could be a cyclic subgroup, generated by a , of \mathbb{F}_p^* , the multiplicative group of the finite field \mathbb{F}_p , where the prime p is given in binary. Testing if $b \in \langle a \rangle$, for $b \in \mathbb{F}_p^*$ is considered computationally hard. The search version of solving for x such that $a^x = b$ is the *discrete log* problem, widely believed intractable for random primes p .

Cayley Table Representation. The Cayley table representation of G , in contrast, makes the CGM problem algorithmically easy: we can define a graph $X = (V, E)$ with $V = G$ as vertex set and $(x, y) \in E$ if $xs = y$ or $ys = x$ for a generator $s \in S$. Then, g is in the subgroup generated by S if and only if the vertex g is reachable from the identity element of G . Indeed, this is an instance of undirected graph reachability which has a polynomial time and even a *deterministic logspace* algorithm due to Reingold [27]. That is, CGM is in the complexity class L (which is contained in P). Since it is in L, it is also in the circuit complexity class $\text{AC}(\log n) = \text{AC}^1$, which means it has log-depth polynomial-size circuits of unbounded fanin. Equivalently, this means CGM has a logarithmic time CRCW PRAM algorithm (we will define the relevant parallel complexity classes in Section 2). Henceforth, we will assume the groups to be given by their multiplication table. In this setting, linear time algorithm for abelian CGM is also known [22].

Parallel Complexity of CGM and Group Isomorphism. Chattopadhyay, Torán and Wagner [7] have shown that the Group Isomorphism problem of checking if two groups G_1 and G_2 given as input by their multiplication tables are isomorphic can be solved by quasipolynomial size constant-depth circuits. While the question whether or not Group Isomorphism is in P is open and is intensely studied in recent times [30, 15, 14], the above parallel complexity upper bound implies that even Parity is not reducible to Group Isomorphism! Similarly, Fleischer has observed, based on [7] that the CGM problem can also be solved by quasipolynomial size constant depth circuits. Since there is no hardness result for CGM, pinpointing its parallel complexity is an interesting question.

As already mentioned, Barrington et al [6] have made nice progress showing that CGM for abelian groups is in $\text{AC}(\log \log n)$. Indeed, since the resulting circuits are dlogtime uniform, the upper bound is FOLL (which means first-order definable with $\log \log n$ quantifier depth,

where n is the size of the group). Further, they also show that CGM for nilpotent groups is in the class $AC((\log \log n)^2)$ and CGM for solvable groups of class d are in $AC(d \log \log n)$. The interesting questions in the static setting is to improve these upper bounds and/or extend these results to other classes of groups.

In this paper we study the *dynamic parallel complexity* of CGM for abelian groups. Before we describe our results, we give some background.

Dynamic complexity. Dynamic algorithms, broadly, deals with the design of efficient algorithms for problems when the input is modified with small changes. The aim is to solve the problem, for the modified input, significantly more efficiently than running the best known “static” algorithm from scratch. The measure of efficiency is crucial here and defines the model of computation. Dynamic algorithms is a burgeoning field of research (see e.g. [17] and [21, 11, 25]) with many applications that require handling large inputs subject to small changes over time.

From a *parallel complexity perspective*, we have the framework of Patnaik and Immerman [26] that is rooted in descriptive complexity [20]. Closely related is the work of Dong, Su, and Topor [12]. Here the ideal solution is to obtain a dynamic algorithm for the considered problem that runs in *constant parallel time*. Theoretically, constant parallel time is $O(1)$ time on a CRCW PRAM model (denoted by $CRCW(1)$, where the CRCW model is the most liberal as it allows for concurrent reads and writes). It is well-known that this coincides with the complexity class AC^0 (the class of problems solvable by constant-depth boolean circuits). From a descriptive complexity perspective, when the circuits are dlogtime uniform (more details in Section 2) this corresponds to FO, the class of problems expressible in first-order logic. The dynamic complexity class DynFO [26] is the class of problems for which there exist FO update formula that have access to constantly many auxiliary relations, such that after small changes to the problem input, the formula correctly computes the output of the problem as well as updates to the auxiliary relations. These different ways of describing $O(1)$ parallel time are essentially equivalent: because FO and *uniform* AC^0 are equivalent [5]. There is renewed interest in this model of computation since a long-standing open problem, whether directed graph reachability is in DynFO, under single edge changes [26], was resolved in the affirmative [8].

In the present paper, it is more convenient to describe our results, which are essentially algorithmic and do not have a logical flavour, in terms of the parallel class $CRCW(1)$ (or equivalently circuit class AC^0).

The results of this paper. In this paper, we obtain results on the dynamic parallel complexity of abelian CGM and abelian Group Isomorphism. Our motivation is to see if we can exploit the underlying group structure to give a dynamic $CRCW(1)$ algorithm for the CGM membership queries while the generating set S is dynamically changing with insertions and deletions. We are able to obtain for the abelian group case the following results.

- First, we consider the more general problem of Monoid Membership, where G is a *monoid* input by its multiplication table. When G is a *commutative monoid* we give a deterministic dynamic $CRCW(1)$ algorithm for membership testing that supports $O(1)$ insertions and deletions in each step. The algorithm requires a one-time SAC^1 preprocessing step. The main idea is to maintain the monoid M in a tree-like data structure. The cyclic monoids are at the leaves of the tree and each internal node has the submonoid of M generated by set of all its descendant leaves. Furthermore, each internal node will also hold submonoids corresponding to deletions of its descendant nodes.

4:4 Dynamic Complexity of Abelian CGM

- We can use this tree-like data structure more powerfully in the case of abelian groups to obtain a randomized dynamic CRCW(1) algorithm for abelian CGM that supports $\text{polylog}(|G|)$ insertions/deletions to S in each step. The algorithm needs a one-time CRCW($\log n$) ($O(\log n)$ time algorithm with polynomially many parallel processors) preprocessing step. The main fact that we exploit here is that adding $\text{polylog}(n)$ many unary numbers can be done in CRCW(1). Thus, from an abelian subgroup H given by $\text{polylog}(n)$ many generators we can randomly sample from H and hence list out all of H with high probability in CRCW(1).
- If the number of insertions/deletions is at most $O(\log n / \log \log n)$ then we obtain a deterministic dynamic CRCW(1) algorithm for abelian CGM which needs a one-time CRCW($\log n$) preprocessing step. Here our techniques are linear algebra based: we need to consider some *miniature* linear algebra problems where the number of variables is $O(\log n)$ and we adapt existing linear algebraic techniques to solve this.
- We obtain analogous results for the dynamic abelian Group Isomorphism. We also consider sublinearly many modifications to the Cayley table as well in all the cases.

The techniques used. The main dynamic complexity technique used is the idea of *muddling* [9]. Imagine that for a dynamic problem we have an CRCW(1) algorithm \mathcal{A} that after each small changes answers queries and updates the auxiliary data structures in $O(1)$ time. And it uses a AC^1 preprocessing step at the beginning to setup the auxiliary data structures. However with successive updates, the auxiliary data structures gradually deteriorate, to the point that after $\log n$ change steps the algorithm can no longer answer the queries and the auxiliary data structures are rendered ineffective. The muddling technique implies that such a problem is in fact in dynamic CRCW(1), meaning that there exists an CRCW(1) algorithm that answers the queries after each small change, and updates the auxiliary data structures, and does so for arbitrarily long sequence of changes. This broad technique is applied in this paper to the problems considered and we will have occasion to see the muddling technique in detail.

In addition, we crucially use the structure of finite abelian groups, and some tree-like data structures. We are also able to use reductions of the CGM problem to some linear algebra problems, such that in the dynamic setting we can utilize efficient matrix inverse and determinant updates under small rank changes to the matrices.

Organization. In Section 2 we give some basic definitions and notation, and some background about the dynamic parallel complexity model. In Section 3 we explain the dynamic CRCW(1) algorithm for commutative monoid membership under single insertions/deletions to the generating set. Sections 4 and 5 contain, respectively, the randomized and deterministic dynamic CRCW(1) algorithms for abelian CGM. In Section 6 we apply the CGM results to obtain dynamic CRCW(1) algorithms for Abelian Group Isomorphism. Finally, in Section 7 we discuss dealing with small changes to the group multiplication table itself. In the interest of space, proofs are pushed to the Appendix. Lemma statements in the main are hyperlinked to their proofs in the Appendix.

2 Preliminaries

Group Theory. A *monoid* (M, \cdot) is a set M equipped with a binary operation \cdot , that is associative and has an identity element. (N, \cdot) is called a submonoid of (M, \cdot) if $N \subseteq M$ containing the identity element and is closed under the binary operation \cdot . A monoid whose

binary operation is commutative is called a *commutative monoid*. A monoid is called a *group* if all its elements have inverses, i.e., for each monoid element there exists another element such that their product gives the identity. A commutative group is called an *Abelian Group*.

Complexity Classes. We will mainly consider parallel complexity classes defined by boolean circuits. Let $AC(t(n))$ denote the class of decision problems that have polynomial-size circuits of *depth* $t(n)$ for inputs of size n , where the AND and OR gates of the circuit are allowed to be unbounded fanin. This circuit model is essentially equivalent to $t(n)$ parallel time on a CRCW PRAM model (with polynomially many processors, denoted by $CRCW(t(n))$), where CRCW allows for concurrent reads and writes to a memory location. More details of these connections can be found in [20]. In particular, $AC(1)$ is usually denoted AC^0 and AC^1 denotes $AC(\log n)$. The class $AC(\log \log n)$ is of interest in this paper due to the result of Barrington et al [6] showing that abelian CGM is in uniform $AC(\log \log n)$. This class is also denoted FOLL in [6] (for first-order formulas with $\log \log n$ depth quantifiers for size n inputs). We use both notations interchangeably. If we restrict the circuits to be monotone (negation allowed at the input gates) and fanin of AND gates to be constant in the above circuit families, then the corresponding complexity classes are denoted by $SAC(t(n))$. For example, SAC^0 and SAC^1 circuit families are same as AC^0 and AC^1 respectively, except that the AND gates can have only a constant size fanin and the negations are allowed only at the input gates [31].

An $AC(t(n))$ algorithm will actually be given by a family of circuits $\{C_n\}_{n>0}$, where C_n solves the problem for inputs of length n , has depth $t(n)$, and size bounded by some polynomial n^c for a constant $c > 0$. We need a *uniformity condition* that tells us how efficiently we can construct the circuits C_n . A stringent condition is the so-called *dlogtime uniformity*: Each gate in the circuit C_n can be described using $O(\log n)$ bits and the uniformity condition requires that the gate connections can be checked in deterministic time linear in $O(\log n)$ by a random access machine [5]. The class dlogtime uniform AC^0 coincides with FO, where the structures on which the formulas are evaluated are equipped with some suitable predicates [5].

The parallel dynamic algorithms in this paper are describable by circuits that are dlogtime uniform.

The parallel dynamic complexity model. We briefly explain the parallel dynamic complexity model. The underlying model for describing the algorithms can be seen as a CRCW PRAM. That means the algorithm can use polynomially many parallel processors accessing a shared memory that allows concurrent reads and concurrent writes with well defined notion of which write succeeds. We can also give a circuit complexity description for the model.

1. For each problem there is a well-defined notion of small changes to the input.
2. In the CRCW PRAM setting, the algorithm uses n^c processors for length n inputs for some constant $c > 0$ that depends on the problem.
3. At each time instant the algorithm receives as input $i(n)$ small changes to the problem input. With respect to the problem input at time instant t , the algorithm is required to output the answer in constant time. I.e., within time instant $t + O(1)$.
4. In the boolean circuit setting, for inputs of length n the model can be seen as a layered boolean circuit that is of width n^c for length n inputs, where the layers denote the time instants. At each layer it receives as input the $i(n)$ changes. For the input at layer t it needs to output the answer before layer $t + O(1)$.

5. We use DynAC^0 to broadly denote the class of problems that have dynamic algorithms that take $O(1)$ time with polynomially many processors². We explicitly state the number of small changes to the input that can be handled at each time step. We also refer to such dynamic algorithms as DynAC^0 algorithms, or equivalently as dynamic CRCW(1) algorithms.

Depending on the problem at hand, the dynamic algorithm usually works by creating a suitable data structure from the given input which it updates with the small changes to the input.

3 A Dynamic CGM Algorithm for Commutative Monoids

In this section, we consider the more general problem of *Cayley Monoid Membership* for commutative monoids: Given a commutative monoid M by its multiplication table, a subset $S \subseteq M$, and an element $m \in M$, check if m is in the submonoid $\langle S \rangle$ generated by S . By abuse of notation, we term this the CGM problem for commutative monoids.

We present a tree-based data structure to maintain the generating set S , using which we obtain a DynAC^0 algorithm that supports a single insertion/deletion to/from the subset S at each step. We will use this data structure with suitable modifications in Section 4.

The CGM problem for monoids. It is known that CGM for monoids is reducible to directed graph reachability. To see this just construct the Cayley digraph $\text{Cay}(M, S)$ of the monoid M corresponding to generating set S . The graph has vertex set M and for every $m \in M$ and every $m' \in S$, the directed edge $(m, m \cdot m')$ is in the edge set. Clearly, an element $t \in M$ is in the submonoid $\langle S \rangle$ iff there is a directed path from the monoid identity e to t in this digraph. Hence Cayley Membership for monoids is in NL. We note here that, though we know dynamic reachability to be in DynAC^0 under constantly many changes, this reduction of the monoid membership problem to reachability doesn't directly give a DynAC^0 bound for the monoid membership problem. This is because, even after the insertion/deletion of a single element to/from S , the graph $\text{Cay}(M, S)$ changes drastically, i.e., $O(n)$ many edges are affected. We only know how to handle $\text{polylog}(n)$ many changes in DynAC^0 even for the undirected reachability problem.

So, we will use the weaker upper bound of SAC^1 for the CGM problem for commutative monoids. This upper bound actually gives a tree-like data structure, using which we obtain the DynAC^0 algorithm for the problem that supports single insertions/deletions to S .

As M is commutative, any element of the submonoid $\langle S \rangle$ is expressible as a product of powers of elements in S : $\prod_{s \in S} s^{e_s}$, where $0 \leq e_s \leq n$ and $n = |M|$.

We claim that the entire submonoid $\langle S \rangle$ can be listed as the output of an SAC^1 circuit. The circuit takes S as input, as a n -bit binary number with i^{th} bit indicating whether the element $m_i \in M$ is in S , and outputs an n -bit binary number whose i^{th} bit is 1 iff $m_i \in M$ is in the submonoid $\langle S \rangle$.

Let $S_1, S_2 \subseteq M$ such that the monoid identity 1 is in both S_1 and S_2 . Their product is defined as $S_1 \cdot S_2 = \{a \cdot b \mid a \in S_1, b \in S_2\}$.

► **Proposition 1.** *Given as input the subsets S_1 and S_2 of a monoid M their product $S_1 \cdot S_2$ can be computed in SAC^0 .*

² This coincides with DynFO [26] when the dlogtime uniformity conditions are met.

For each $a \in S$ let $P_a = \{a^i : 1 \leq i \leq n\}$. Notice that P_a can be computed directly from the multiplication table for M in L which is contained in SAC^1 .

The tree-like data structure for S . We create a balanced binary tree T with leaves labelled by the distinct elements $a \in S$. Let the root of T be ρ . To the leaf labelled a , we associate the submonoid P_a . With each internal node τ , we associate a commutative submonoid M_τ of M , inductively defined as the product $M_\tau = M_\mu \cdot M_\nu$, where μ and ν are its two children. Since $|M| = n$, the tree T has depth bounded by $\log n$. By Proposition 1 the tree T can be created by an SAC^1 circuit. Moreover, for each internal node of T , the following is immediate.

► **Proposition 2.** *For each node τ of the tree T the subset M_τ associated with τ is the commutative submonoid generated by the subset $L_\tau = \{a \in S \mid a \text{ such that } \tau \text{ has the leaf labelled } a \text{ as descendant}\}$. I.e., $M_\tau = \langle L_\tau \rangle$.*

It is clear that the subset associated with root ρ of the tree T , is actually $\langle S \rangle$. Thus, we have:

► **Lemma 3.** *The CGM problem for commutative monoids is in SAC^1 .*

The Dynamic Setting. We will modify the above construction to obtain a dynamic data structure. First we expand the tree T , by including leaves for every element of M . For each element $a \in M$, we pre-compute its power set P_a as already defined. However, we will make P_a to be the submonoid associated with the leaf labelled a , $M_a = P_a$, precisely if $a \in S$, and otherwise we associate the identity element 1, i.e., $M_a = \{1\}$.

We will need to dynamically maintain the following data at each node of the expanded tree. Only the data associated with the tree nodes change with the changing generating set S , while the tree structure itself remains unchanged.

1. For each node ν of the tree we have the submonoid M_ν generated by the subsets associated with the leaves below node ν in the tree.
2. Additionally, at each node ν we will maintain the submonoid $M_{\nu:\mu}$ for each descendant μ of ν in the tree, where $M_{\nu:\mu}$ denotes the submonoid generated by the subsets associated with all leaves that are descendants of ν but are not descendants of μ . Equivalently, it is as if the submonoid associated with node μ is reset to $\{1\}$ and then the rest of the submonoid M_ν is computed.

Essentially, as in Propositions 1 and 2, given a subset S we can construct the tree data structure along with the data at each node as described above.

► **Proposition 4.** *Given as input a subset $S \subseteq M$ for a commutative monoid M (given by its multiplication table), we can construct the tree data structure T along with the data at each node as described above in SAC^1 . Furthermore, given a membership query $m \in M$, testing if m is in the current submonoid $\langle S \rangle$ can be done in AC^0 .*

Handling single insertions and deletions. Next we show that the above data structure supports single insertions and deletions to S at each time step. The updates to the data structure can be carried out in AC^0 as described below.

First we consider deletions. Suppose $a \in S$ is deleted. Then we need to update for each tree node ν , the submonoid M_ν associated with it and the submonoids $M_{\nu:\mu}$ for each of its descendant μ . Clearly, M_ν remains unchanged if a is not a descendant of ν . Similarly, $M_{\nu:\mu}$ also remains unchanged if a is a descendant of μ . In general, the required updates

4:8 Dynamic Complexity of Abelian CGM

are explicated by the Algorithm 1 (which contains the description of both delete and insert updates colour coded as red and blue respectively). Similarly, in the case of insertion of an element $a \in M$ to S , we need to update the submonoids associated with the tree nodes.

■ Algorithm 1 Delete(a).	Insert(a).
1 $M_a \leftarrow \{1\}$	$M_a \leftarrow P_a$
2 for $\nu \in V(T)$ such that $a \in \text{descendants}(\nu)$ do in parallel	
3 $M_\nu \leftarrow M_{\nu:a}$	$M_\nu \leftarrow M_\nu \cdot P_a$
4 for $\mu, \nu \in V(T)$ such that μ and a are descendants of ν do in parallel	
5 $\mu' \leftarrow \text{LCA}(\mu, a)$	
6 if $\mu' \neq \mu$ then	
7 Let μ_1, μ_2 be the children of μ' such that a and μ are descendants of μ_1 and μ_2 respectively	
8 $M_{\nu:\mu} \leftarrow (M_{\nu:\mu'}) \cdot (M_{\mu_1:a}) \cdot (M_{\mu_2:\mu})$	$M_{\nu:\mu} \leftarrow (M_{\nu:\mu'}) \cdot (M_{\mu_1} \cdot P_a) \cdot (M_{\mu_2:\mu})$

As each node in the tree can be processed in parallel and each requires just a product of a constant number of pre-computed submonoids (lines 3 and 8), we have the following

► **Proposition 5.** *The above deletion and insertion operations can be carried out in AC^0 .*

The correctness of the update procedures in Algorithm 1 follows by induction on the validity of the tree data structure. The inductive hypothesis being that the data structure contains valid information about the various submonoids at the time step just before the insert/delete updates. To summarize the above results, we have the main theorem of this section.

► **Theorem 6.** *Let M be a commutative monoid given by its multiplication table and $S \subseteq M$ generate a submonoid $\langle S \rangle$ of M . Then there is a deterministic DynAC^0 algorithm that answers membership queries $m \in \langle S \rangle$ given $m \in M$ and supports single insertions and deletions to the generating set S at each time step, which requires a one-time SAC^1 preprocessing step for initialization of auxiliary data structures.*

4 The Dynamic CGM Problem for Abelian Groups

Let G be an n -element abelian group given as input by its multiplication table. Let S be a subset of G . We want to maintain a structure that supports efficient membership testing in the subgroup $H = \langle S \rangle$ generated by S . That means efficiently supporting the following operations.

1. Given a query element $g \in G$ test if $g \in H = \langle S \rangle$ and, if so, express g as a product of the generators in S .
2. The dynamic version of the problem requires that we efficiently support insertions/deletions to set S . We have seen how to handle, even in the setting of commutative monoids, single insertions or deletions to S at each step. More generally, we would like to handle bulk insertions and deletions at each step.

Preprocessing for dynamic abelian CGM. We will first obtain a generating set of size at most $\log n$ for G using which we can represent all elements of G .

An *independent* generating set [16, Section 3.2] for an abelian group G is a generating set $\{g_1, g_2, \dots, g_\ell\}$ such that $g_1^{e_1} \cdot g_2^{e_2} \cdots g_\ell^{e_\ell} = 1$ for $0 \leq e_i \leq o(g_i) - 1$ if and only if $e_i = 0$, where $o(g_i)$ is the order of g_i for each i . As a consequence of independence, every element $g \in G$ is *uniquely expressible* as $g = g_1^{e_1} \cdot g_2^{e_2} \cdots g_\ell^{e_\ell}$ for $0 \leq e_i \leq o(g_i) - 1$. It is easy to see that $\ell \leq \log |G|$. The next proposition easily follows from [16, Theorem 3.2.2].

► **Proposition 7.** *Given as input a finite abelian group G by its multiplication table, an independent generating set for G (of size at most $\log n$) can be computed in $\text{AC}(\log n)$.*

Thus, in a one-time preprocessing step, we can compute an independent generating set $\{g_1, g_2, \dots, g_\ell\}$ from the multiplication table of the input group G as well as the unique expression for each $g \in G$ as $g = g_1^{e_1} \cdot g_2^{e_2} \cdots g_\ell^{e_\ell}$ for $0 \leq e_i \leq o(g_i) - 1$. The two preprocessing steps for G are summarized below.

1. For each pair $(g, i), g \in G, 0 \leq i \leq \ell - 1$, we compute and store the power g^i in an $n \times n$ table. This can be done in a straightforward $\text{AC}(\log n)$ computation, since the product of two elements in the table is computable in AC^0 . In particular, this computation also yields the order $o(g)$ of each $g \in G$.
2. By Proposition 7, in $\text{AC}(\log n)$ we compute for G an independent generating set $T = \{g_1, g_2, \dots, g_t\}$, $t \leq \log n$ and also a representation for each $g \in G$ as a product $g = \prod_{i=1}^t g_i^{e_i}$.

Additionally, we note the easy consequence of Barrington et al's FOLL algorithm [6] for abelian CGM.

► **Lemma 8.** *Let $S \subseteq G$, for an abelian group G given by its multiplication table as input. In FOLL a $\log |G|$ size subset T of S can be computed that generates the same subgroup as S .*

Randomized DynAC⁰ Algorithm for Abelian CGM

We now present a randomized DynAC⁰ algorithm for maintaining the subgroup $H = \langle S \rangle$ of G , given by its generating set. More precisely, the algorithm can process $\text{polylog}(n)$ insertions and deletions per step and answer membership queries to the group $\langle S \rangle$ in $O(1)$ parallel time³. The following is a crucial lemma that is used by the query algorithm in this section.

► **Lemma 9.** *Let $T \subseteq G$ be of size at most $\log^c n$ for some constant c , where G is an abelian group, $|G| = n$, given by its multiplication table with independent generating set $G = \langle g_1, g_2, \dots, g_\ell \rangle$. Then in randomized AC^0 we can list out the subgroup $\langle T \rangle$ generated by T . In particular, membership testing in the subgroup generated by T can be done in randomized AC^0 .*

Given Lemma 9, we can focus on just maintaining a $\text{polylog}(n)$ size generating set for the subgroup $\langle S \rangle$ while it undergoes $\text{polylog}(n)$ many insertions/deletions in a step.

Auxiliary data structure. First we will create a variant of the tree-based data structure described in Section 3 to represent the generating set S and certain subgroups of $\langle S \rangle$. We will continually update this data structure, as we process the bulk insertions and deletions that occur at each time step.

³ Updates to the auxiliary data structures are in deterministic AC^0 , while queries require randomization.

4:10 Dynamic Complexity of Abelian CGM

1. Let S be the current generating set and $2^{k-1} < |S| \leq 2^k$, for positive integer $k \leq \lceil \log n \rceil$. To begin with, we create an $O(\log n)$ depth *full binary tree* B , with 2^k leaves. We associate, for each generator $x \in S$, the cyclic subgroup $\langle x \rangle$ with a distinct leaf of B . The remaining leaves are associated with the trivial subgroup $\{1\}$. As it is a full binary tree, its nodes can be indexed by $1, 2, \dots, 2^{k+1} - 1$ with 1 as index for the root. For each $i > 1$, the node indexed i has as parent the node indexed $\lfloor i/2 \rfloor$.
2. Let τ be an internal node of the tree B with children μ and ν . Inductively, to each internal node τ we associate the subgroup, $H_\tau = H_\mu \cdot H_\nu$ which is the product of the subgroups H_μ and H_ν that are associated with the two children. Notice that the product $H_\mu \cdot H_\nu$ is indeed a subgroup of G as G is abelian. Letting S_τ denote the set of leaves in S below node τ . Then, notice that $H_\tau = \langle S_\tau \rangle$ for each node τ of the tree. The root is labeled with $H = \langle S \rangle$.
3. Additionally, for each internal node τ and for each *descendant* μ of τ we keep the subgroup generated by $S_\tau \setminus S_\mu$, which we denote by $H_{\tau:\mu}$. Thus, at each node τ we have the list of subgroups $H_{\tau:\mu}$ with generating set $S_\tau \setminus S_\mu$, one for each descendant μ of τ .
4. Finally, using Lemma 8 we compute, in FOLL, $\log n$ size generating sets $T_{\tau\mu} \subseteq S_\tau \setminus S_\mu$ for each subgroup $H_{\tau:\mu}$ at each node τ in parallel. Similarly, for subgroup H_τ associated with each internal node τ , a $\log n$ size generating set $T_\tau \subseteq S_\tau$ is computed.
5. For access to the data maintained at each node in the tree, we will have an array of pointers indexed by $1, \dots, 2^{k+1} - 1$. Furthermore, we will keep a boolean array $A[i, j], 1 \leq i, j \leq 2^{k+1} - 1$ where $A[i, j] = 1$ if and only if i is an ancestor of j .

The following lemma is immediate.

► **Lemma 10.** *The tree data structure B , for the generating set S can be built in $O(\log n)$ parallel time (i.e. in $AC(\log n)$).*

The tree data structure B useful in Section 5 as well.

Handling bulk insertions and deletions. At any point of time during the computation, the current generating set S , is maintained as a data structure described above. Then changes to the generating set arrive in the form of two sets I and D for insertions and deletions respectively, such that $|I|, |D| = O(\log^{c+1} n)$. The actual generating set then becomes $S \cup I \setminus D$. We will first show that a membership query occurring at this point of time can be answered in $O(1)$ parallel time.

► **Lemma 11.** *Given the data structure for S along with the update sets of insertions I and deletions D , each of $\log^{c+1} n$ size, we can test if some group element g is in the subgroup generated by $(S \cup I) \setminus D$ in AC^0 .*

Continual rebuilding of the tree data structure B . This is the crucial part of the dynamic algorithm⁴. Let's assume that at time instant t , we have the tree data structure for the current generating set, denoted by $S^{(t)}$, available (this assumption is particularly valid at the starting time instant because of the preprocessing). At this instant, the new insertion and deletion bulk updates, denoted by $I^{(t)}$ and $D^{(t)}$ respectively, arrive. Using Lemma 11, we can answer membership queries about the subgroup generated by $S^{(t+1)} = (S^{(t)} \cup I^{(t)}) \setminus D^{(t)}$ in $O(1)$

⁴ This continual rebuilding of data structure is a modified adaptation of the muddling technique from [9].

time. In fact, we can do this for any time instant $t + i$ ($1 \leq i < \log n$), by using Lemma 11, keeping the generating set S to be $S^{(t)}$ and I and D to be the insertions and deletions respectively accumulated so far from time instant t to $t + i$, since I and D are still bounded by $\text{polylog}(n)$ ($|\cup_{j=1}^i I^{(t+j)}|, |\cup_{j=1}^i D^{(t+j)}| \leq i \log^c n < \log^{c+1} n$ since $i < \log n$). But, we can't keep doing this forever, as eventually the accumulated I and D grow beyond $\text{polylog}(n)$. In a sense, the tree data structure is only useful for $\log n$ rounds. To remedy this, we will start AC^1 computation for building the tree data structure for the subgroup generated by $(S^{(t)} \cup I^{(t)}) \setminus D^{(t)}$ at the time instant $t + 1$ using Lemma 10 and the result of this computation will be available at time instant $t + \log n$. After this point we can continue to use Lemma 11 for another $\log n$ rounds to answer the CGM queries. In fact, at each time instant $t + i$ we will have an AC^1 computation thread started up for building the tree data structure for the subgroup generated by $S^{(t+i)}$. At any time instant there are at most $\log n$ such thread running and hence the number of gates in any layer remains polynomially bounded.

To summarize, we have shown the following theorem.

► **Theorem 12.** *There is a randomized DynAC^0 algorithm for the abelian Cayley Group Membership problem, CGM, that requires a one-time AC^1 preprocessing step, and supports $\text{polylog}(n)$ insertions and deletions to the generating set.*

5 A Deterministic Dynamic Algorithm for Abelian CGM

We now present a *deterministic* DynAC^0 algorithm for abelian CGM that can process bulk insertions/deletions of size $t = O(\frac{\log n}{\log \log n})$. The algorithm is linear algebraic. We first observe a property of abelian groups in terms of prime factorization of their order. Let G be an n -element abelian group given by its multiplication table. Let $n = p_1^{a_1} \times p_2^{a_2} \times \dots \times p_\mu^{a_\mu}$ be its prime factorization, where p_i are distinct primes. By the structure of finite abelian groups [16, Theorem 3.3.1] $G = G_1 \times G_2 \times \dots \times G_\mu$ is a direct product where G_i is the p_i -Sylow subgroup of G .⁵

For a subset $S \subseteq G$ consider the subgroup $H = \langle S \rangle$ generated by S . Let $b_i = n/p_i^{a_i}$, $1 \leq i \leq \mu$ and $S_i = \{x^{b_i} \mid x \in S\}$, $1 \leq i \leq \mu$. Then $H_i = \langle S_i \rangle$ is a subgroup of G_i for each i and $H = H_1 \times H_2 \times \dots \times H_\mu$.

An element $g \in G$ is in the subgroup H if and only if $g^{b_i} \in H_i$ for each i . As a consequence we can reduce the dynamic abelian CGM problem to the dynamic abelian CGM problem for abelian p -groups. We state this as a lemma.

► **Lemma 13.** *Given an n -element abelian group G by its Cayley table, by a one-time preprocessing computation in AC^1 we can compute Cayley tables for each Sylow subgroup G_i . Furthermore, all powers of elements of G can be pre-computed and stored in an array. Hence the parallel dynamic complexity of abelian CGM maintaining S , supporting say $t(n)$ insertions/deletions at each step is AC^0 reducible to the same problem for abelian p -groups.*

Now, we can focus on the CGM problem restricted to abelian p -groups only.

Reduction to integer linear equations. Using the Proposition 7 we can pre-compute an independent generating set $\{g_1, g_2, \dots, g_\ell\}$ for the abelian p -group G . Let $|G| = n = p^m$ and $o(g_i) = p^{m_i}$ for each i , where $m_1 + m_2 + \dots + m_\ell = m$, and $\ell \leq \log n$. We also

⁵ Let G be a finite group of order $n = p_1^{a_1} \times p_2^{a_2} \times \dots \times p_\mu^{a_\mu}$. Then for each i , G has at least one subgroup of order $p_i^{a_i}$, which is known as a p_i -Sylow subgroup of G . However, if G is abelian then there is a unique p_i -Sylow subgroup of G which we can denote by G_i . In that case, $G = G_1 \times G_2 \times \dots \times G_\mu$.

4:12 Dynamic Complexity of Abelian CGM

have $m = \log_p n \leq \log n$. Each $g \in G$ has a unique representation (pre-computed) as $g = \prod_{i=1}^{\ell} g_i^{b_i}$, $0 \leq b_i \leq p^{m_i} - 1$. Thus g can also be represented as an ℓ -dimensional integer column vector $\bar{b} = [b_1 b_2 \dots b_{\ell}]^T$.

We will dynamically maintain a logarithmically bounded subset T of the generating set S such that T generate the same group as S , i.e., $|T| = O(\log n)$ and $\langle S \rangle = \langle T \rangle$. As explained, we will represent elements of T by ℓ -dimensional column vectors. Thus, $g \in \langle T \rangle$ iff the system of integer linear equations $Ax = \bar{b}$ is feasible, where the matrix A has columns corresponding to each generator in T , and the i^{th} row of the system of equations is computed modulo p^{m_i} for $1 \leq i \leq \ell$. We can suitably scale each equation to get a system of integer linear equations modulo p^m . Since p^m is a composite for $m > 1$, the usual recipe for feasibility of linear equations based on matrix rank does not directly apply. However, we can rewrite $Ax = \bar{b} \pmod{p^m}$ equivalently as integer linear equations

$$Ax + p^m y = \bar{b} \tag{1}$$

where y is a column vector of ℓ new variables. Letting $[A|p^m I_{\ell}] = \tilde{A}$, and $z = [x|y]^T$ we can write this as $\tilde{A}z = \bar{b}$, noting that \tilde{A} is full row rank. Thus, the CGM problem for abelian p -groups is reduced to finding integer solution to Equation (1). To test the feasibility of such system of equations, we have the following lemma.

► **Lemma 14** ([3], Theorem 3.13). *For a prime p , the system $\tilde{A}z = \bar{b}$ (and hence $Ax = \bar{b} \pmod{p^m}$) is feasible (i.e., has an integer solution) iff GCD of the $\ell \times \ell$ subdeterminants of \tilde{A} and the GCD of the $\ell \times \ell$ subdeterminants of the augmented matrix $[\tilde{A}|\bar{b}]$ have the same highest power of p dividing them both.*

Since the matrix A in Equation (1) has $\ell \leq \log n$ rows and $t \leq 2 \log n$ columns the number of $\ell \times \ell$ subdeterminants is polynomially bounded. So, if we could compute the determinant of each $\ell \times \ell$ submatrix in $\text{AC}(\log \log n)$, we can execute the above feasibility test as well in $\text{AC}(\log \log n)$, and hence solve the CGM problem. Indeed, we have the following lemma to compute determinant of such small matrices.

► **Lemma 15.** *Let A be a square matrix of dimension $\text{polylog}(n)$ with entries that are polynomially bounded in n , then each bit of $\det(A)$ can be computed in $\text{AC}(\log \log n)$.*

We will need to dynamically maintain the determinants and inverses of all the $\ell \times \ell$ submatrices of $\tilde{A} = [A|p^m I_{\ell}]$. The following lemma shows that, at least in the beginning, all these can be precomputed in $\text{AC}(\log \log n)$, giving way for an AC^0 algorithm for CGM (part 3).

► **Lemma 16.** *Let $Ax = \bar{b} \pmod{p^m}$ be a system of integer linear equations modulo p^m , where $p^m = n$ is input in unary, and $A \in \mathbb{Z}^{\ell \times t}$ and $\bar{b} \in \mathbb{Z}^{\ell}$, $t = O(\log n)$.*

1. *Let $\tilde{A} = [A|p^m I_{\ell}]$. We can compute the determinants of all square submatrices of \tilde{A} and $[\tilde{A}|\bar{b}]$ in $\text{AC}(\log \log n)$ (i.e. in $\log \log n$ parallel time).*
2. *Furthermore, for the nonsingular submatrices we can also compute their inverses in $\text{AC}(\log \log n)$.*
3. *Given the above we can test the feasibility of $Ax = \bar{b} \pmod{p^m}$ and solve for x in AC^0 .*

This preprocessing of the matrix \tilde{A} needs to be combined with a variant of the matrix inverse lemma stated below [18] (this is a variant of the so-called Sherman-Morrison-Woodbury formula) to dynamically compute solutions to $\tilde{A}z = \bar{b}$. This formula essentially allows for a quick updation of the data computed using Lemma 16 for A , if A is replaced with $A + A'$ for a small rank matrix A' .

► **Lemma 17** (Binomial Matrix Theorem [18]). *Let M be an invertible $r \times r$ matrix over any field (or ring). Let C , U and V be $t \times t$, $r \times t$ and $t \times r$ matrices respectively, over the same field/ring. The inverse of $M + UCV$ is $M^{-1} - M^{-1}U(I + CVM^{-1}U)^{-1}CVM^{-1}$ if $M + UCV$ is invertible.*

Similarly to update determinants quickly we will need the following.

► **Lemma 18** (Matrix Determinant Lemma). *If M is a $r \times r$ matrix over a field and U and V are $r \times t$ and $t \times r$ matrices then $\det(M + UV^T) = \det(I_t + V^T M^{-1}U) \det(M)$.*

Finally, we will also require the following lemma to compute the small matrix inverses and determinants, viz. $(I + CVM^{-1}U)^{-1}$ and $\det(I_t + V^T M^{-1}U)$ respectively, required by Lemmas 17 and 18.

► **Lemma 19** ([10], Theorem 8).

1. *Let $t = O(\frac{\log n}{\log \log n})$ and B be a $t \times t$ integer matrix with entries bounded by p^m and q be an $O(\log \log n)$ bit prime number. Then both $\det(B) \pmod{q}$ and B^{-1} over \mathbb{F}_q can be computed in AC^0 .*
2. *Furthermore, by Chinese remaindering, $\det(B)$ and hence B^{-1} , if it exists, can both be computed in AC^0 by applying the first part for several distinct primes q_i and different submatrices.*

We will now see how to use these in the context of processing $O(\log n / \log \log n)$ bulk insertions and deletions.

Processing Bulk Insertions. Suppose \hat{T} is the set of insertions to S , where $|\hat{T}| = t = O(\log n / \log \log n)$. Thus, the system of linear equations $Ax = \bar{b} \pmod{p^m}$ is now modified to $[A|\hat{A}]x = \bar{b} \pmod{p^m}$, where the columns of A correspond to T , the new columns \hat{A} correspond to the insertions \hat{T} , and \bar{b} is the integer vector corresponding to a $g \in G$ whose membership we want to test in $\langle T \cup \hat{T} \rangle$.

We note that in the modified linear equation above, the original coefficient matrix has been modified in at most t columns. Thus, Lemmas 17 and 18 are applicable. With these we can update the data computed by Lemma 16 in AC^0 . It can be recomputed in AC^0 by Lemma 19 as at most $O(\log n / \log \log n)$ columns are modified in any submatrix, thinking of the new columns as modifications of zero columns. Furthermore the recomputations involves computing the determinant and inverse of matrices of dimension at most $t = O(\log n / \log \log n)$, where those matrices have integer entries given as input in unary (because each of them is at most p^m in magnitude). A crucial difficulty in the application of Lemmas 17 and 18 is that a submatrix M of A , whose inverse/determinant we need to update, may itself not be invertible. We can deal with this by maintaining the data computed by Lemma 16 for the *invertible* matrices $\xi I - M$, for all submatrices M of A , where ξ is an indeterminate. Lemma 15 and parts 1 and 2 of Lemma 16 can be applied *mutatis mutandis* to matrices $\xi I - M$ (for the submatrices M of \hat{A}). The determinant of $\xi I - M$ will be a degree r polynomial in ξ and $(\xi I - M)^{-1}$ will have entries that are rational functions $f(\xi)/g(\xi)$ where f and g are of degree at most r , where $r = O(\log n)$. Consider Lemma 17 and Lemma 18 applied to $\xi I - M$ instead of M . Notice that $\det(M + UV^T)$ is the constant term of $\det(M + UV^T - \xi I)$ which we can compute in AC^0 , essentially by Lemma 19. Similarly, by Lemma 19 the inverse $(M + UCV^T)^{-1}$, if it exists, can be computed in AC^0 from Lemma 17 applied to $\xi I - M$.

Processing Bulk Deletions. Let \hat{T} be the set of elements that are deleted from S . It is clear that modifications to the matrix A are required only if some elements from T are deleted, i.e., $T \cap \hat{T} \neq \emptyset$. However, deletions are bit trickier than insertions to handle. This is because, we can't simply drop the columns from A corresponding to $\hat{T} \cap T$ after bulk deletion \hat{T} , as it need not be the case that what remains in T is still a generating set for the $\langle S \setminus \hat{T} \rangle$. We might possibly need to include some elements from $S \setminus \hat{T}$ to $T \setminus \hat{T}$ for getting a correct small generating set for $S \setminus \hat{T}$. The number of columns to be dropped from A is clearly bounded by $O(\log n / \log \log n)$. However, with respect to number of columns that are to be included in A after the change, Lemmas 17 and 18 enable us to update the submatrices' inverse and determinant only if this number is bounded by $O(\log n / \log \log n)$. To our relief, it is indeed the case. We show the following about finite abelian p -groups in general. For a finite abelian p -group G , $|G| = p^m$, let $S \subseteq G$. Let T be a subset of S such that $\langle S \rangle = \langle T \rangle$. Without loss of generality, we assume that $|S| \geq 2$. Then we have the following.

► **Lemma 20.** *Let G be a finite abelian p -group, $|G| = p^m$, and $T \subseteq S \subseteq G$ such that $\langle T \rangle = \langle S \rangle$. Then for any $g \in T$, there is an $h \in S \setminus T$ such that, $\langle S \setminus \{g\} \rangle = \langle (T \setminus \{g\}) \cup \{h\} \rangle$.*

Proof. Let $H = \langle T \rangle$ and $K = \langle T \setminus \{g\} \rangle$. Since the groups are abelian $H = K \langle g \rangle$. Suppose $|H| = p^{m_1}$ and $|K| = p^{m_2}$. Let $\mu = m_1 - m_2$. Then $|H|/|K| = p^\mu$, which is the number of distinct cosets of K in H . Furthermore, as H/K is cyclic, generated by Kg , it follows that $Kg \in H/K$ is an element of order p^μ . Hence, we have the disjoint union $H = \sqcup_{j=0}^{p^\mu-1} Kg^j$.

▷ **Claim 21.** For $j \geq 0$ and α relatively prime to p , the subgroups $K \langle g^{p^j} \rangle$ and $K \langle g^{p^j \alpha} \rangle$ are identical.

To see this it suffices to note that for any finite cyclic group $\langle a \rangle$ of order d and any α relatively prime to d , a^α is also a generator of the cyclic group $\langle a \rangle$. Hence, the cyclic subgroups $\langle g^{p^j} \rangle$ and $\langle g^{p^j \alpha} \rangle$ are identical which proves the claim.

For each $s \in S$, let Kg^{e_s} be the coset to which it belongs. Writing $e_s = p^{\ell_s} \alpha_s$ for α_s relatively prime to p , the above claim implies the subgroups $K \langle s \rangle$ and $K \langle g^{p^{\ell_s}} \rangle$ are identical. That means the two subgroups $\langle T \cup \{s\} \setminus \{g\} \rangle$ and $\langle T \cup \{g^{p^{\ell_s}}\} \setminus \{g\} \rangle$ are identical.

Now, among elements in $S \setminus T$, let h be an element with the least ℓ_h . We claim that $\langle S \setminus \{g\} \rangle = \langle T \cup \{h\} \setminus \{g\} \rangle$.

Suppose $s \in S \setminus \{g\}$ is some other element. Then $\ell_s \geq \ell_h$, by the choice of h . That means $K \langle g^{p^{\ell_s}} \rangle$ is a subgroup of $K \langle g^{p^{\ell_h}} \rangle$, which implies $s \in K \langle g^{p^{\ell_h}} \rangle = K \langle h \rangle = \langle (T \setminus \{g\}) \cup \{h\} \rangle$, completing the proof. ◀

From Lemma 20, it is clear that after deletion of \hat{T} , there is a $O(\log n / \log \log n)$ size set $R \subseteq (S \setminus \hat{T})$ such that $\langle (T \setminus \hat{T}) \cup R \rangle = \langle S \setminus \hat{T} \rangle$. Hence, after the deletion of \hat{T} from S if we can find such a set R , then we can update the matrix A by dropping the columns corresponding to the elements in \hat{T} and including columns corresponding to R so that the modified matrix A in Equation (1) correctly corresponds to the CGM question with respect to the modified generating set S . For the modified matrix, we can update for each $\ell \times \ell$ submatrix its inverse and determinant in AC^0 using binomial matrix theorem and matrix determinant lemma (Lemmas 17 and 18). This is because the determinants and inverses involved in Lemmas 17 and 18 can be computed in AC^0 due to Lemma 19 as their size bounded by $O(\log n / \log \log n)$.

But, we can't afford to search for R in $S \setminus \hat{T}$, which is of $O(n)$ size. However, we can make use of the tree data structure B , of Section 4 that essentially maintains a $\text{polylog}(n)$ size generating set for S . That is, given B , we always have a $\text{polylog}(n)$ size subset P of S

available, such that it generates the same group as S . We can exhaustively search for a valid R in P , since the search space is polynomially bounded ($(O(\log n / \log \log n))^{\text{polylog}(n)} = \text{poly}(n)$). For each choice of R we can use the AC^0 routine for handling $O(\log n / \log \log n)$ insertions to the generating set, to find the span $\langle (T \setminus \hat{T}) \cup R \rangle$. On top of that, we can find an R' such that $\langle (T \setminus \hat{T}) \cup R' \rangle$ contains all the elements that could be generated with any other choice of R . With R' , we finally update $T \setminus \hat{T}$ to become $(T \setminus \hat{T}) \cup R'$ and update all the relevant matrix inverses and determinants.

Continual rebuilding of data structure. We note that the data required by the algorithm to answer membership queries in AC^0 are:

- A generating set T of size $O(\log n)$ such that $\langle T \rangle = \langle S \rangle$.
- A $\text{polylog}(n)$ size generating set P of $\langle S \rangle$ such that $P \subseteq S$.
- An $\ell \times |T|$ dimensional matrix A corresponding to the generating set. Determinant and inverses of all non-singular $\ell \times \ell$ submatrices of $[A|p^m I_\ell]$.

We have already seen in Section 4 that $\text{polylog}(n)$ size generating set P can be maintained in DynAC^0 even under $\text{polylog}(n)$ size bulk changes with help of the tree data structure, though it requires a one time AC^1 preprocessing step.

Having initialized the above auxiliary data structure, after every bulk insertion/deletion we can update them in AC^0 at least for $\log \log n$ time steps. Within this time period, the small generating set T remains logarithmically bounded, because even though after every time step, size of T could grow by $O(\log n / \log \log n)$, it can only grow by $O(\log n)$ over $\log \log n$ steps. Also, within this time period, we can check feasibility of $Ax = \bar{b} \pmod{p^m}$ in AC^0 and hence answer membership queries. We use the muddling technique to extend this $\log \log n$ window such that we can, for arbitrary long sequence change-steps, answer the CGM queries in AC^0 after each bulk insertion/deletion step.

In the beginning we have the correct small generating set $T_0 = T$, because of initialization. After the first change step, say $i = 1$, we start an $\text{AC}(\log \log n)$ computation thread for computing an at most $\log n$ size subset T_1 of the modified T such that it generates the same subgroup. The result of this thread, T_1 is available after a delay of $\log \log n$ time steps, i.e., at time step $i = \log \log n + 1$. With respect to T_1 and the insertion/deletions that accumulate during the recomputation phase, we have all the updated information about the corresponding matrix A , i.e., the determinants and inverses of all $\ell \times \ell$ submatrices of A . So, when the next bulk insertion/deletion arrives, we can update the auxiliary information again in AC^0 . In fact, we start an $\text{AC}(\log \log n)$ recomputation thread for computing the small generating set T_i at every time step i within the $\log \log n$ time window so that we have logarithmically bounded generating set at time step $i + \log \log n$. Still the number of recomputation threads running at any given time step is $O(\log \log n)$, thus keeping the overall circuit size polynomially bounded.

To summarize we have shown the following theorem.

► **Theorem 22.** *There is a deterministic dynamic AC^0 algorithm for the abelian Cayley Group Membership problem, CGM, that supports $O(\log n / \log \log n)$ insertions and deletions to the generating set, and requires a one-time AC^1 preprocessing step.*

6 Dynamic Abelian group isomorphism

Let G_1 and G_2 be abelian groups, each given a multiplication table as input, say T_1 and T_2 , respectively. Let $S_1 \subseteq G_1$ and $S_2 \subseteq G_2$ be subsets. In the static setting, there is a simple polynomial time algorithm for checking if $\langle S_1 \rangle$ and $\langle S_2 \rangle$ are isomorphic: it suffices to list

out the two subgroups $\langle S_1 \rangle$ and $\langle S_2 \rangle$, check they have the same order n , and check for each factor k of n that the number of elements of order k in the two subgroups $\langle S_1 \rangle$ and $\langle S_2 \rangle$ is the same.⁶

In Section D we give a DynAC^0 algorithm for *dynamic* abelian group isomorphism that supports insertions and deletions to both S_1, S_2 . Thus, we have shown the following.

► **Theorem 23.** *There is a randomized (respectively deterministic) DynAC^0 algorithm for abelian group isomorphism that supports $\text{polylog}(n)$ (respectively $O(\log n / \log \log n)$) insertions and deletions at each step to the generating sets of the two groups.*

7 Making the multiplication table dynamic

We have assumed so far that the overall group G (or monoid) is unchanged and only the generating set for the CGM problem is dynamic. Suppose now that the entries of the multiplication table of G can be modified dynamically. When the table's entries change, it may no longer represent a group (or a monoid). The binary operation $* : G \times G \rightarrow G$ is just a *magma*, in general. However, we can show that the dynamic algorithms for abelian CGM still hold, with the proviso that the membership query answers are correct only when the magma is actually an abelian group.

The main property we use here is that at most one group has its multiplication table within linear (i.e. $O(n)$) edit distance from the multiplication table of an n -element magma G .⁷ Moreover, from the magma multiplication table we can *decode* this unique group in AC^0 . We note that Ergün et al [13] have shown stronger results for this problem in the context of spot checkers; they give randomized self-correction algorithms for a variety of problems. However, for a self-contained presentation, we include a simple proof of a weaker Lemma 28 yielding:

► **Theorem 24.** *There is a randomized DynAC^0 algorithm that supports $O(n / \log n)$ changes to the multiplication table and $\text{polylog}(n)$ insertions/deletions to the generating set, with the proviso that when the multiplication table decodes to an abelian group the membership queries are answered with respect to it, and when it does not decode to an abelian group then the query answers could be incorrect. There is also a deterministic DynAC^0 algorithm that supports $O(n / \log n)$ changes to the multiplication table and $\log n / \log \log n$ insertions/deletions to the generating set, with the same proviso as described above.*

8 Conclusion and open ends

We address the dynamic complexity of CGM and isomorphism problems for finite abelian groups input by their multiplication table under $O(\log n / \log \log n)$ changes to the generating set while also allowing sublinear changes to the table itself to be in constant parallel time with an initial logarithmic parallel time precomputation. We can also handle $\text{polylog}(n)$ changes to the generating set by allowing randomness. For the more general algebraic structures, namely commutative monoids, we gave a foundational method to handle single changes, on which the preceding are built.

Natural open questions are to extend the results to more general groups like nilpotent and solvable groups.

⁶ Two finite abelian groups are isomorphic iff for each positive integer k the number of elements of order k in the two groups coincide [16].

⁷ By the edit distance between multiplication tables $op_1 : G \times G \rightarrow G$ and $op_2 : G \times G \rightarrow G$ we mean the number of pairs $(a, b) \in G \times G$ such that $op_1(a, b) \neq op_2(a, b)$.

References

- 1 Miklós Ajtai. \sum^1_1 -formulae on finite structures. *Ann. Pure Appl. Log.*, 24(1):1–48, 1983. doi:10.1016/0168-0072(83)90038-6.
- 2 Miklós Ajtai. Approximate counting with uniform constant-depth circuits. In *Advances In Computational Complexity Theory, Proceedings of a DIMACS Workshop, New Jersey, USA, December 3-7, 1990*, pages 1–20, 1990. doi:10.1090/DIMACS/013/01.
- 3 Vikraman Arvind and T. C. Vijayaraghavan. Classifying problems on linear congruences and abelian permutation groups using logspace counting classes. *Comput. Complex.*, 19(1):57–98, 2010. doi:10.1007/S00037-009-0280-6.
- 4 László Babai, Eugene M. Luks, and Ákos Seress. Permutation groups in NC. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 409–420. ACM, 1987. doi:10.1145/28395.28439.
- 5 David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1 . In *Proceedings: Third Annual Structure in Complexity Theory Conference, Georgetown University, Washington, D. C., USA, June 14-17, 1988*, pages 47–59, 1988.
- 6 David Mix Barrington, Peter Kadau, Klaus-Jörn Lange, and Pierre McKenzie. On the complexity of some problems on groups input as multiplication tables. *Journal of Computer and System Sciences*, 63(2):186–200, 2001. doi:10.1006/JCSS.2001.1764.
- 7 Arkadev Chattopadhyay, Jacobo Torán, and Fabian Wagner. Graph isomorphism is not AC^0 -reducible to group isomorphism. *ACM Trans. Comput. Theory*, 5(4):13:1–13:13, 2013. doi:10.1145/2540088.
- 8 Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. Reachability is in DynFO. *J. ACM*, 65(5):33:1–33:24, 2018. doi:10.1145/3212685.
- 9 Samir Datta, Anish Mukherjee, Thomas Schwentick, Nils Vortmeier, and Thomas Zeume. A strategy for dynamic programs: Start over and muddle through. *Log. Methods Comput. Sci.*, 15(2), 2019. doi:10.23638/LMCS-15(2:12)2019.
- 10 Samir Datta, Anish Mukherjee, Nils Vortmeier, and Thomas Zeume. Reachability and distances under multiple changes. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 120:1–120:14, 2018. doi:10.4230/LIPICs.ICALP.2018.120.
- 11 Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. Parallel batch-dynamic graphs: Algorithms and lower bounds. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1300–1319, 2020. doi:10.1137/1.9781611975994.79.
- 12 Guozhu Dong, Jianwen Su, and Rodney W. Topor. Nonrecursive incremental evaluation of datalog queries. *Ann. Math. Artif. Intell.*, 14(2-4):187–223, 1995. doi:10.1007/BF01530820.
- 13 Funda Ergün, Sampath Kannan, Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan. Spot-checkers. *J. Comput. Syst. Sci.*, 60(3):717–751, 2000. doi:10.1006/JCSS.1999.1692.
- 14 Joshua A. Grochow and Youming Qiao. On p-group isomorphism: Search-to-decision, counting-to-decision, and nilpotency class reductions via tensors. In Valentine Kabanets, editor, *36th Computational Complexity Conference, CCC 2021, July 20-23, 2021, Toronto, Ontario, Canada (Virtual Conference)*, volume 200 of *LIPICs*, pages 16:1–16:38. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.CCC.2021.16.
- 15 Joshua A. Grochow and Youming Qiao. On the complexity of isomorphism problems for tensors, groups, and polynomials I: tensor isomorphism-completeness. In James R. Lee, editor, *12th Innovations in Theoretical Computer Science Conference, ITCS 2021, January 6-8, 2021, Virtual Conference*, volume 185 of *LIPICs*, pages 31:1–31:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ITCS.2021.31.
- 16 M. Hall. *The Theory of Groups*. AMS Chelsea Publishing Series. AMS Chelsea Pub., 1999.
- 17 Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms - A quick reference guide. *ACM J. Exp. Algorithmics*, 27:1.11:1–1.11:45, 2022. doi:10.1145/3555806.

- 18 H.V. Henderson and S.R. Searle. On deriving the inverse of a sum of matrices. *SIAM Review*, 23(1):53–60, 1981.
- 19 William Hesse, Eric Allender, and David A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *J. Comput. Syst. Sci.*, 65(4):695–716, 2002. doi:10.1016/S0022-0000(02)00025-9.
- 20 Neil Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999. doi:10.1007/978-1-4612-0539-5.
- 21 Giuseppe F. Italiano, Silvio Lattanzi, Vahab S. Mirrokni, and Nikos Parotsidis. Dynamic algorithms for the massively parallel computation model. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*, pages 49–58, 2019. doi:10.1145/3323165.3323202.
- 22 T. Kavitha. Linear time algorithms for abelian group isomorphism and related problems. *Journal of Computer and System Sciences*, 73(6):986–996, 2007. doi:10.1016/J.JCSS.2007.03.013.
- 23 Meena Mahajan and V. Vinay. Determinant: Combinatorics, algorithms, and complexity. *Chic. J. Theor. Comput. Sci.*, 1997, 1997. URL: <http://cjtcs.cs.uchicago.edu/articles/1997/5/contents.html>.
- 24 Pierre McKenzie and Stephen A. Cook. The parallel complexity of abelian permutation group problems. *SIAM J. Comput.*, 16(5):880–909, 1987. doi:10.1137/0216058.
- 25 Krzysztof Nowicki and Krzysztof Onak. Dynamic graph algorithms with batch updates in the massively parallel computation model. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2939–2958, 2021. doi:10.1137/1.9781611976465.175.
- 26 Sushant Patnaik and Neil Immerman. Dyn-fo: A parallel, dynamic complexity class. *J. Comput. Syst. Sci.*, 55(2):199–209, 1997. doi:10.1006/JCSS.1997.1520.
- 27 Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17:1–17:24, 2008. doi:10.1145/1391289.1391291.
- 28 Á. Seress. *Permutation Group Algorithms*. Cambridge Tracts in Mathematics. Cambridge University Press, 2003.
- 29 Charles C. Sims. Computation with permutation groups. In Stanley R. Petrick, Jean E. Sammet, Robert G. Tobey, and Joel Moses, editors, *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation, SYMSAC 1971, Los Angeles, California, USA, March 23-25, 1971*, pages 23–28. ACM, 1971. doi:10.1145/800204.806264.
- 30 Xiaorui Sun. Faster isomorphism for p-groups of class 2 and exponent p. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023*, pages 433–440, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3564246.3585250.
- 31 H. Venkateswaran. Circuit definitions of nondeterministic complexity classes. *SIAM J. Comput.*, 21(4):655–670, 1992. doi:10.1137/0221040.

A Missing proofs from Section 3

► **Proposition 2.** For each node τ of the tree T the subset M_τ associated with τ is the commutative submonoid generated by the subset $L_\tau = \{a \in S \mid a \text{ such that } \tau \text{ has the leaf labelled } a \text{ as descendant}\}$. I.e., $M_\tau = \langle L_\tau \rangle$.

Proof. This is easily proved by induction on the tree T . The point to note is that commutativity of the monoid M is crucial: if $M_\mu = \langle L_\mu \rangle$ and $M_\nu = \langle L_\nu \rangle$ by induction hypothesis, then we note that $M_\mu \cdot M_\nu = \langle L_\mu \cup L_\nu \rangle$ because all the elements commute with each other. Hence $M_\tau = M_\mu \cdot M_\nu = \langle L_\tau \rangle$. ◀

► **Lemma 3.** The CGM problem for commutative monoids is in SAC¹.

Proof. Given $m \in M$, to check if $m \in \langle S \rangle$ we just need to check if m is in the submonoid M_ρ associated with the root ρ of T . ◀

► **Proposition 4.** *Given as input a subset $S \subseteq M$ for a commutative monoid M (given by its multiplication table), we can construct the tree data structure T along with the data at each node as described above in SAC¹. Furthermore, given a membership query $m \in M$, testing if m is in the current submonoid $\langle S \rangle$ can be done in AC⁰.*

Proof. For the tree construction, it suffices to observe that we can do the computation of each $M_{\nu:\mu}$ in parallel in SAC¹. For membership testing, if the ρ is the root of the tree then the submonoid $M_\rho = \langle S \rangle$ is available as a list at the node ρ . Hence membership testing is in AC⁰. ◀

B Missing proofs from Section 4

► **Lemma 8.** *Let $S \subseteq G$, for an abelian group G given by its multiplication table as input. In FOLL a $\log |G|$ size subset T of S can be computed that generates the same subgroup as S .*

Proof. Let $S = \{x_1, x_2, \dots, x_s\}$. For each $i > 1$ in parallel, we can check if x_{i+1} is in $\langle x_1, x_2, \dots, x_i \rangle$ using the FOLL algorithm of [6]. If $x_{i+1} \notin \langle x_1, x_2, \dots, x_i \rangle$ then we include x_{i+1} into the set T . Clearly, $|T| \leq \log |G|$ and generates the same subgroup and S . ◀

► **Lemma 9.** *Let $T \subseteq G$ be of size at most $\log^c n$ for some constant c , where G is an abelian group, $|G| = n$, given by its multiplication table with independent generating set $G = \langle g_1, g_2, \dots, g_\ell \rangle$. Then in randomized AC⁰ we can list out the subgroup $\langle T \rangle$ generated by T . In particular, membership testing in the subgroup generated by T can be done in randomized AC⁰.*

Proof. Let $T = \{x_1, x_2, \dots, x_r\}$. For each $g \in G$ we have the pre-computed unique product

$$g = \prod_{i=1}^{\ell} g_i^{\alpha_i},$$

using the independent generating set $\{g_1, g_2, \dots, g_\ell\}$. In particular, for each $x_j \in T$ we have

$$x_j = \prod_{i=1}^{\ell} g_i^{\alpha_{ij}},$$

where $0 \leq \alpha_{ij} \leq o(g_i) - 1$ for each $i \in [\ell]$. As explained below, we can randomly sample from the group generated by T by picking numbers $\beta_j \in_R [n], 1 \leq j \leq r$ uniformly at random and computing the product

$$x = \prod_{j=1}^r x_j^{\beta_j}.$$

The number of such products is n^r . Furthermore, each element of the subgroup $\langle T \rangle$ occurs in this product with multiplicity exactly $|\{(\beta_1, \beta_2, \dots, \beta_r) \mid \prod_j x_j^{\beta_j} = 1\}|$, as this set is the kernel of the group homomorphism mapping $(\beta_1, \beta_2, \dots, \beta_r) \mapsto \prod_{j=1}^r x_j^{\beta_j}$. Thus, x is uniformly distributed in $\langle T \rangle$. If we draw, say n^2 such samples x in parallel, the probability that all elements of $\langle T \rangle$ appear is at least $1 - e^{-n}$. Finally, we analyze the complexity of computing the product $x = \prod_{j=1}^r x_j^{\beta_j}$. Notice that it amounts to computing the product $\prod_{i=1}^{\ell} g_i^{\sum_{j=1}^r \beta_j \alpha_{ij}}$.

Now, each of these ℓ exponents $\sum_{j=1}^r \beta_j \alpha_{ij}$ is a $\log^c n$ sum of *unary* numbers and can be computed modulo the unary number $o(g_i)$ in AC^0 . The final product can be looked up in the pre-computed table to find x . This proves that the group $\langle T \rangle$ can be listed in randomized AC^0 and hence membership testing in $\langle T \rangle$ is also in randomized AC^0 . ◀

► **Lemma 10.** *The tree data structure B , for the generating set S can be built in $O(\log n)$ parallel time (i.e. in $\text{AC}(\log n)$).*

Proof. The tree has $\log n$ levels and the straightforward computation required at each of the (at most $|S|$) nodes at each level is AC^0 . ◀

► **Lemma 11.** *Given the data structure for S along with the update sets of insertions I and deletions D , each of $\log^{c+1} n$ size, we can test if some group element g is in the subgroup generated by $(S \cup I) \setminus D$ in AC^0 .*

Proof. Let $D = \{x_{i_1}, x_{i_2}, \dots, x_{i_d}\}$ be the deletions from S , $d = \log^{c+1} n$. From the data structure for S we can find the d subtrees rooted at nodes $\nu_{i_1}, \nu_{i_2}, \dots, \nu_{i_d}$ where each node ν_{i_j} is the root of the maximal subtree that has exactly one deletion x_{i_j} occurring among its leaves. This can be done in $O(1)$ parallel time using the ancestor boolean array $A[u, v]$. In the data structure for S we already have a $\log n$ size generating set, say $T_{\nu_{i_j} x_{i_j}}$, for each subgroup $H_{\nu_{i_j} : x_{i_j}}$.

Additionally, we find the *maximal subtrees*, rooted at nodes $\mu_{\ell_1}, \mu_{\ell_2}, \dots, \mu_{\ell_s}$ of the tree, such that these subtrees contain no deletion x_{i_j} as descendant. To get a bound on s , note that each such maximal subtree has as sibling a subtree that contains one or more deletions among its leaves. The roots of all subtrees that have deletions among its leaves are just the ancestors of the ν_{i_j} nodes, and hence are at most $d \lceil \log n \rceil$ in number (each leaf has at most $\lceil \log n \rceil$ ancestors). Thus, $s = O(d \log n) = O(\log^{c+2} n)$. At each node μ_{ℓ_j} ($j \in [s]$) we already have a $\log n$ size generating set, $T_{\mu_{\ell_j}}$ for each subgroup $H_{\mu_{\ell_j}}$. Let $T = \cup_j T_{\nu_{i_j} x_{i_j}}$ and $\hat{T} = \cup_j T_{\mu_{\ell_j}}$. It follows from the above that we can compute T and \hat{T} in $O(1)$ parallel time.

Putting it all together, the group generated by $(S \cup I) \setminus D$ is actually generated by $\hat{T} \cup T \cup I$ which is of $\text{polylog}(n)$ size. Therefore, applying Lemma 9 we can do membership testing in this subgroup in randomized AC^0 . ◀

C Missing proofs from Section 5

► **Lemma 15.** *Let A be a square matrix of dimension $\text{polylog}(n)$ with entries that are polynomially bounded in n , then each bit of $\det(A)$ can be computed in $\text{AC}(\log \log n)$.*

Proof. It is well known that the determinant of a matrix of n variables can be computed by boolean threshold circuits⁸ of polynomial size and logarithmic depth, i.e. it is in TC^1 . (Proof sketch: the determinant of a matrix of polynomial dimension with polynomial in n bit entries can be computed in arithmetic SAC^1 [23, Table 2]. In other words, it can be computed by a layered logarithmic depth circuit with gates from $\{+, -, *\}$ where the $*$ -gates have fan-in 2. Now by applying [19, Corollary 6.7] each layer of this arithmetic circuit can be simulated in TC^0 , i.e. constant-depth threshold circuits). Hence, replacing n with $\log n$, it follows that the determinant of matrices of $\text{polylog}(n)$ dimension with $\text{polylog}(n)$ bit entries can be computed by a threshold circuit of depth $O(\log \log n)$ and size $\text{polylog}(n)$. Furthermore, threshold gates of $\text{polylog } n$ fanin can be computed by $\text{poly}(n)$ size uniform AC^0 [1, 2]. Replacing the threshold gates by the corresponding AC^0 -circuit of size $\text{poly}(n)$ completes the proof. ◀

⁸ Threshold circuits allow for unbounded fanin threshold gates apart from NOT, AND, and OR gates.

► **Lemma 16.** *Let $Ax = \bar{b} \pmod{p^m}$ be a system of integer linear equations modulo p^m , where $p^m = n$ is input in unary, and $A \in \mathbb{Z}^{\ell \times t}$ and $\bar{b} \in \mathbb{Z}^l$, $t = O(\log n)$.*

1. *Let $\tilde{A} = [A|p^m I_\ell]$. We can compute the determinants of all square submatrices of \tilde{A} and $[\tilde{A}|\bar{b}]$ in $\text{AC}(\log \log n)$ (i.e. in $\log \log n$ parallel time).*
2. *Furthermore, for the nonsingular submatrices we can also compute their inverses in $\text{AC}(\log \log n)$.*
3. *Given the above we can test the feasibility of $Ax = \bar{b} \pmod{p^m}$ and solve for x in AC^0 .*

Proof. For the first part, the number of square submatrices is polynomially bounded as \tilde{A} has dimension $O(\log n) \times O(\log n)$. Reducing modulo p^m , the entries of the matrix are bounded by n . Thus, by Lemma 15 it follows that the determinant as an integer can be computed in $\text{AC}(\log \log n)$. Reducing modulo p^m yields the answer and we know from [19] that the division by a unary number is possible in AC^0 .

For the second part, consider every nonsingular submatrix N , i.e. $\det N$ is non-zero. We can compute the entries of N^{-1} by Cramer's rule, as each cofactor of N is also a submatrix of \tilde{A} . Since we can compute division of $O(\log n)$ -bit integers in AC^0 (see [19, Theorem 5.1]) it follows that these computations can also be done in $\text{AC}(\log \log n)$.

For the last part, notice that feasibility can be tested in AC^0 , given the data of first two parts, by Lemma 14. From the set of all $l \times l$ non-singular submatrices of \tilde{A} , we choose N , that has the least power of p dividing its determinant, $\nu_p(\det(N))$ is the smallest.

W.l.o.g. N is the first l columns of \tilde{A} . Let $Nz' = \bar{b}$ be the system of equations obtained by keeping only the first l columns of \tilde{A} and truncating z beyond the l th coordinate. The solution to this is $z' = N^{-1}\bar{b}$, where the right hand side is computable in AC^0 given N^{-1} . Now we can extend this solution to a solution of $\tilde{A}z = \bar{b}$ by putting $z_i = z'_i$ for each column i in $[l]$ and $z_i = 0$ for other columns. It is easy to see that this must be a solution of $\tilde{A}z = \bar{b}$, as we know the latter is feasible. However, this solution may not necessarily be integer, as we would like. But this rational solution will have the property that denominators of the solution are relatively prime to p as we show below.

Because of Cramer's rule, for any $i \in [l]$, $z_i = \det([N_i|\bar{b}]) / \det(N)$, where N_i is the matrix obtained from N by dropping its i th column. Let s be the gcd of the determinants of all $l \times l$ submatrices of \tilde{A} . Let t be the gcd of the determinants of all $l \times l$ submatrices of $[\tilde{A}|\bar{b}]$. From the feasibility criterion of Lemma 14, we have that $\nu_p(s) = \nu_p(t)$. But, from the choice of N it is clear that, $\nu_p(s) = \nu_p(\det(N))$ and hence $\nu_p(\det(N)) = \nu_p(t)$. In particular, this implies that, $\nu_p([N_i|\bar{b}]) \geq \det(N)$. From this, we have that the denominator of z_i is relatively prime to p for any $i \in [l]$.

Now, we can uniquely modify such a solution to be integral easily in AC^0 . ◀

D Missing parts from Section 6

Let G_1 and G_2 be abelian groups, each given a multiplication table as input, say T_1 and T_2 , respectively. Let $S_1 \subseteq G_1$ and $S_2 \subseteq G_2$ be subsets. In the static setting, there is a simple polynomial time algorithm for checking if $\langle S_1 \rangle$ and $\langle S_2 \rangle$ are isomorphic: it suffices to list out the two subgroups $\langle S_1 \rangle$ and $\langle S_2 \rangle$, check they have the same order n , and check for each factor k of n that the number of elements of order k in the two subgroups $\langle S_1 \rangle$ and $\langle S_2 \rangle$ is the same.⁹

⁹ Two finite abelian groups are isomorphic iff for each positive integer k the number of elements of order k in the two groups coincide [16].

We give a DynAC⁰ algorithm for the *dynamic version* of abelian group isomorphism that supports insertions and deletions to both S_1 and S_2 . Let $n_1 = \prod_{x \in S_1} o(x)$ and $n_2 = \prod_{y \in S_2} o(y)$, where the orders $o(x), o(y), x \in S_1, y \in S_2$ can be pre-computed for the elements of the two groups G_1 and G_2 . Let $n_1 = \prod_{i=1}^r p_i^{a_i}$ and $n_2 = \prod_{i=1}^r p_i^{b_i}$ be their prime factorizations. We can assume both n_1 and n_2 have the same prime factors. Otherwise, $\langle S_1 \rangle$ and $\langle S_2 \rangle$ are not isomorphic. Let $n_{1i} = n_1/p_i^{a_i}$ and $n_{2i} = n_2/p_i^{b_i}$ and $S_{1i} = \{x^{n_{1i}} \mid x \in S_1\}$ and $S_{2i} = \{y^{n_{2i}} \mid y \in S_2\}$ for $1 \leq i \leq r$. Since a finite abelian group is a direct product of its (unique) Sylow subgroups we have

► **Proposition 25.** *The groups $\langle S_1 \rangle$ and $\langle S_2 \rangle$ are isomorphic iff their p_i -Sylow subgroups $\langle S_{1i} \rangle$ and $\langle S_{2i} \rangle$ are isomorphic.*

Thus, as argued in Sections 4 and 5, it suffices to solve the problem for abelian p -groups. Henceforth, we assume both $\langle S_1 \rangle$ and $\langle S_2 \rangle$ are p -groups. The following lemma from McKenzie and Cook's work [24], paraphrased in our context, is useful for our algorithm.

► **Lemma 26** ([24], Proposition 6.4). *Let $\langle S_1 \rangle \leq G_1$ and $\langle S_2 \rangle \leq G_2$ be abelian p -groups, and k be largest positive integer such that $p^k \leq \max\{|G_1|, |G_2|\}$. For $1 \leq j \leq k$ let $S_{1j} = \{x^{p^j} \mid x \in S_1\}$ and $S_{2j} = \{y^{p^j} \mid y \in S_2\}$. Then $\langle S_1 \rangle$ and $\langle S_2 \rangle$ are isomorphic if and only if $|\langle S_{1j} \rangle| = |\langle S_{2j} \rangle|$ for $1 \leq j \leq k$.*

Effectively, the above lemma is a reduction from abelian group isomorphism to abelian CGM. Thus, as observed in [7], in the static setting we can note that the above lemma immediately shows that abelian group isomorphism problem we consider can be solved by AC(log log n) circuits with majority gates. This is by applying the Barrington et al algorithm [6] to enumerate the subgroups $\langle S_{1j} \rangle$ and $\langle S_{2j} \rangle$ in AC(log log n) for each $1 \leq j \leq k$ and then comparing their orders (for which majority gates are required).

Our strategy for the dynamic version is also based on Lemma 26 because we can apply the results for abelian CGM shown in Sections 4 and 5.

In the dynamic setting, where we have insertions and deletions to the generating sets S_1, S_2 , we will use the same data structures developed in Section 4 (for supporting polylog(n) insertions and deletions) and Section 5 (for supporting log n /log log n insertions and deletions) for the abelian CGM problem but now in parallel for all the generating sets $\langle S_{1j} \rangle$ and $\langle S_{2j} \rangle$ for $1 \leq j \leq k$.

In order to compute $|\langle S_{1j} \rangle|$ and $|\langle S_{2j} \rangle|$ from membership queries the following lemma, from [24], is useful.

► **Lemma 27** ([24], Proposition 6.6). *Let $H = \langle g_1, \dots, g_r \rangle$ be a finite abelian p -group. Then, $|H| = t_1 t_2 \dots t_r$ where t_j is the least positive integer such that $g_j^{t_j} \in \langle g_{j+1}, \dots, g_r \rangle$ for $1 \leq j \leq r$.*

In the above lemma, as H is a p -group notice that each t_j is a power of p . We will be applying this lemma to groups $\langle S_{1j} \rangle$ and $\langle S_{2j} \rangle$. As $|\langle S_{1j} \rangle| \leq n_1 \leq |G_1|$ and $|\langle S_{2j} \rangle| \leq n_2 \leq |G_2|$, and both n_1 and n_2 are logarithmic size in binary, for these groups H at most logarithmically many of the integers t_i are more than 1. Letting $t_j = p^{r_j}$, computing the product $\prod_j t_j = p^{\sum_j r_j}$ amounts to adding at most logarithmically many r_j , each logarithmically bounded. As already observed, such tiny additions can be computed in AC⁰.

E Missing parts from Section 7

► **Lemma 28.** *Let M_G denote the multiplication table of the group G . Suppose M is a multiplication table obtained from M_G by changing at most δn many entries of M_G , for $\delta < 1/13$. Then there is an AC⁰ circuit that takes M as input and outputs M_G .*

Proof. For each $x_i \in G$ the row of x_i in M has at most δn errors in it. Thus, the row for the identity element, say $x_1 = e$ is uniquely determined, because $x_1 \in G$ is the unique element with $x_1 x_j = x_j$ for majority of $j \in [n]$.

For each $z \in G$ there is a unique inverse $z^{-1} \in G$ and $zz^{-1} = e = z^{-1}z$. That means in M_G there are exactly n occurrences of e in the multiplication table. Therefore, in M there are at most $(1 + \delta)n$ occurrences of e and at least $(1 - \delta)n$ occurrences of e .

Let $S = \{(z, w) \mid z, w \in G, z * w = e\}$, where $*$ is the product operation in the table M . Then

$$(1 - \delta)n \leq |S| \leq (1 + \delta)n.$$

Thus, for at least $(1 - 2\delta)n$ pairs $(z, w) \in S$ we have $z * w = zw = e$ in G .

Now, in order to recover the correct value of the product $x_i x_j$, we look up the products $(x_i * z) * (w * x_j)$ in the table M . Then we have

- $|\{z \in G \mid x_i * z \neq x_i z\}| \leq \delta n.$
- $|\{w \in G \mid w * x_j \neq wx_j\}| \leq \delta n.$
- $|\{(z, w) \in S \mid z * w \neq zw\}| \leq \delta n.$
- $|\{(z, w) \in S \mid (x_i z) * (wx_j) \neq (x_i z)(wx_j)\}| \leq \delta n.$

Thus, for at least $(1 - 6\delta)n$ pairs $(z, w) \in S$ we have $(x_i * z) * (w * x_j) = (x_i z)(wx_j) = x_i(zw)x_j = x_i x_j$.

If we choose $\delta < 1/13$ then the number of such pairs $(z, w) \in S$ is more than $7n/13$. By approximate majority which can be computed in AC^0 [1, 2], we can find this correct value of $x_i x_j$. We can thus recover the entire table M_G in AC^0 . ◀

The above lemma suggests the following simple dynamic algorithm for the abelian CGM problem that supports $\text{polylog}(n)$ changes to the group multiplication table, as well as bulk insertions/deletions to the generating set (as discussed in Sections 4 and 5).

1. Let M be the current multiplication table. We apply the AC^0 algorithm of Lemma 28 to decode M . Let G be the resulting table.
2. If the decoded table does not give an abelian group, the query answers can be arbitrary (but consistent which can be ensured by remembering the answers to queries already made).
3. Suppose the decoded table gives an abelian group G' . If $G' \neq G$ then for the next $\log n$ steps we rebuild the static data structure for G' in $\text{AC}(\log n)$. We can answer any membership queries, occurring in this window of $\log n$ time steps, arbitrarily. After $\log n$ steps we can replace G with G' and its data structure and continue.

In summary, we have the following.

► **Theorem 24.** *There is a randomized DynAC^0 algorithm that supports $O(n/\log n)$ changes to the multiplication table and $\text{polylog}(n)$ insertions/deletions to the generating set, with the proviso that when the multiplication table decodes to an abelian group the membership queries are answered with respect to it, and when it does not decode to an abelian group then the query answers could be incorrect. There is also a deterministic DynAC^0 algorithm that supports $O(n/\log n)$ changes to the multiplication table and $\log n / \log \log n$ insertions/deletions to the generating set, with the same proviso as described above.*