COUNTING IN UNIFORM TC^0

JUI-LIN LEE

Abstract. In this paper we first give a uniform AC^0 algorithm which uses partial sums to compute multiple addition. Then we use it to show that multiple addition is computable in uniform TC^0 by using *count* only once sequentially. By constructing bit matrix for multiple addition, we prove that multiple product with poly-logarithmic size is computable in uniform TC^0 (by using *count* $k+1$ times sequentially when the product has size $O((\log n)^k)$). We also prove that multiple product with sharply bounded size is computable in uniform AC^0.

1. Introduction

In this paper we study basic counting techniques inside uniform TC^0. We adopt function algebraic approach, for it requires less background and has more mathematical (or at least machine-independent) favor.

The study of complexity classes related to parallel computation is nowadays more important since parallel computing is thought to be useful. In theoretical computer science there are several well-developed parallel models. We will focus on Boolean circuits because remarkable separation results [11],[17] are based on it.

Recall that AC^0 is the class of predicates computable by polynomial size, constant depth, unbounded fan-in circuits with gates AND, OR, NOT. Majority gate is define as follows: $MAJ(x) = 1$ if at least a half bits in the binary string x are 1's, else $MAJ(x) = 0$. TC^0 is the class of predicates computable by polynomial size, constant depth, unbounded fan-in circuits with gates AND, OR, NOT, MAJ.

In the 80's, two important separation results were proved. The first result is the separation of AC^0 and $AC^0(p)$, where p is a prime and $AC^0(m)$ is AC^0 plus modular m counting gates. [9], [1] gave superpolynomial lower bounds for the size of circuits computing parity in AC^0. (Later [18], [11] proved exponential lower bound for parity.) The second result [17], inspired from [16], proved that $AC^0 \subsetneq AC^0(p) \subsetneq AC^0(pq)$, where p, q are distinct primes. However, very little is known beyond $AC^0(pq)$ so far.

What are uniform classes (and why do we study them)? Uniformity means the way by which we construct those circuits. Since it is more difficult to show that a function is computable in class with quite restricted uniformity than in non-uniform class, maybe the uniform condition can be helpful to prove separation results.

A weakness of this attempt is that those results [11],[17] easily imply the separation in uniform cases. (Note that uniform $AC^0 \subsetneq$ non-uniform AC^0 and modular counting gates are quite uniform. Therefore the separation of uniform TC^0 and non-uniform AC^0 trivially separates TC^0 and AC^0 with uniformity.) And so far

Date: August 29, 1997.
1991 *Mathematics Subject Classification.* Primary: 03D15; Secondary: 68D15, 68D25.

we do not know how to use uniformity to either simplify [11], [17], or prove separation on uniform classes beyond $AC^0(pq)$. (Perhaps the only exception is [2].) Even though, since non-uniform approach does not do better recently, it may be worth a while to consider uniform approach. To do so, a better understanding in uniform circuit classes is necessary.

In Section 2 we summarize basic tools (Lemmas 2.5,2.7,2.10,2.13,2.17) in function complexity class A_0 (or function algebra A_0, in convention), which characterizes uniform AC^0. We also introduce function algebra T_0 ($= A_0 + count$), which corresponds to uniform TC^0. (Here $count(x)$ is the number of 1's in the binary string x. Note that $count$ is equivalent to MAJ under AC^0 reduction.) Sections 3,4 are basically formalization of the following: convert numbers into bits, apply $count$ at each column, rewrite results into bits. (The idea is straightforward, and these two sections are merely for completeness.) In Section 5 we give an A_0 algorithm to compute multiple addition by partial sums. This algorithm is new for it is not given explicitly before. In Section 6 we show how one can do partial sum in A_0 if its size is small enough. This avoids the not-constantly many uses of weak multiple addition or sharply bounded counting (that is, count with poly-logarithmic bits). Therefore multiple addition needs to use $count$ only once sequentially. In Section 7 we prove that multiple product with sharply bounded value is computable in A_0. In Section 8 we prove a new result that, by using multiple addition iteratively, multiple product with poly-logarithmic size is computable in T_0. If we define $A_0(count)_k = \{f \in T_0 : f \text{ is defined by using } count \text{ nestedly at most } k \text{ times}\}$, then multiple product $\prod_{i=0}^{n} z(\overrightarrow{x}, i) \leq 2^{p(\log n)}$ is computable in $A_0(count)_{k+1}$ provided that $z \in A_0$, $deg(p) = k$, and n is the size of inputs.

Though all constructions here are based on function algebras A_0, T_0, the author believes that it will not be too difficult for readers to convert the algorithms to their favorite systems.

2. Background

In this section we review some basic results in uniform complexity classes AC^0, TC^0. Instead of using circuit approach, we use function complexity classes A_0, T_0. Following the convention of [6], we call them function algebras. Roughly speaking, a function algebra is the smallest class of functions containing some basic functions and closed under some construction rules. Examples of construction rules are composition, iteration, recursion with some limitation. The advantage of function algebraic approach is that it is machine independent. We will define a function algebra A_0 which characterizes uniform AC^0. (For details see [6].) Then we introduce another function algebra T_0 which corresponds to uniform TC^0. Finally we define $A_0(count)_k$, that is, the class of functions (in T_0) which uses $count$ sequentially at most k many times.

All functions in this paper have domain and codomain $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.

Definition 2.1. $zero(x) = 0$, $s_0(x) = 2x$, $s_1(x) = 2x + 1$, $i_k^n(x_1, \ldots, x_n) = x_k$, $pad(x, y) = 2^{|y|} \cdot x$, $|x| = \lceil \log_2(x + 1) \rceil$, $x\#y = 2^{|x| \cdot |y|}$, $(x)\,mod2 = x - 2 \cdot \lfloor x/2 \rfloor$, $Bit(i, x) = (\lfloor x/2^i \rfloor)\,mod2$, $|x|_2 = ||x||$, $|x|_{k+1} = ||x|_k|$ for $k \geq 2$, $\overrightarrow{x} = x_1, x_2, \ldots, x_m$ means a sequence of natural numbers, and $|\overrightarrow{x}| = max(|x_1|, \ldots, |x_m|)$.

Definition 2.2. Suppose that $h_0(n, \overrightarrow{x}), h_1(n, \overrightarrow{x}) \leq 1$. The function f is defined by CRN (*concatenation recursion on notation*) from g, h_0, h_1 if

$$\begin{aligned} f(0, \overrightarrow{x}) &= g(\overrightarrow{x}), \\ f(s_0(n), \overrightarrow{x}) &= s_{h_0(n,\overrightarrow{x})}(f(n, \overrightarrow{x})) \text{ for } n > 0, \\ f(s_1(n), \overrightarrow{x}) &= s_{h_1(n,\overrightarrow{x})}(f(n, \overrightarrow{x})). \end{aligned}$$

Definition 2.3. A_0 is the smallest class of functions containing the basic functions $zero, s_0, s_1, i^n_k, |x|, \#, Bit(i, x)$, and closed under composition and CRN.

In [12] Immerman developed the notion of first order definability which captures uniform circuits without involving sequential or alternating Turing machines. Because of the robustness of this class, people believe this notion is the right notion of uniform AC^0. In [6] Clote proved that $A_0 = FO$, where FO is one version of uniform AC^0 defined by first order definability.

We will not use this result later, for we will develop all necessary tools directly in A_0.

Definition 2.4. A function f is *sharply bounded* (or *double sharply bounded*) if there is an A_0 function g such that $f(\overrightarrow{x}) \leq |g(\overrightarrow{x})|$ (or $f(\overrightarrow{x}) \leq ||g(\overrightarrow{x})||$).

Let $n = |\overrightarrow{x}|$, then it is easy to show that f is sharply bounded iff $f \leq p(n)$ for some polynomial p, and f is double sharply bounded iff $f \leq c \log n$ for some constant c.

Now we recall some useful results from [3], [7], [8]. Lemmas 2.5,2.7 are from [7]. We give a direct proof of Lemma 2.10 (so that one needs not deal with bounded arithmetic TAC^0). For Lemmas 2.13,2.17, we add remarks to resolve vagueness or gap of their original proofs.

Sharply bounded quantifiers are of the forms $\exists x < |t|$, $\forall x < |t|$. Lemma 2.5 shows that A_0 is closed under sharply bounded quantification.

Lemma 2.5. *If $g, h \in A_0$ and f is defined by*

$$f(x) = \begin{cases} 1 & \textit{if } \exists i \leq |g(x)| \, [h(i, x) = 0], \\ 0 & \textit{else,} \end{cases}$$

then $f \in A_0$.

Proof. See [7]. ☐

Definition 2.6. The function f is defined from g, h by *sharply bounded μ-operator* if

$$f(x) = \begin{cases} i_0 & \text{if } i_0 \leq |g(x)| \wedge h(i_0, x) = 0 \wedge \forall i < i_0 \, (h(i, x) \neq 0), \\ |g(x)| & \text{else.} \end{cases}$$

This is denoted by $f(x) = \mu i < |g(x)| \, [h(i, x) = 0]$.

Since in such case "$h(f(x), x) = 0$?" can be easily checked, we also call it sharply bounded search.

Lemma 2.7. *A_0 is closed under the sharply bounded μ-operator.*

Proof. See [7]. ☐

Definition 2.8.

$$\begin{aligned} x * y &= pad(x,y) + y, \\ Seg(x,i,j) &= \sum_{k=i}^{j} 2^{k-i} Bit(k,x) \text{ for } i < j, \\ MSP(x,j) &= \lfloor x/2^j \rfloor, \\ LSP(x,j) &= x - 2^j \cdot MSP(x,j). \end{aligned}$$

Obviously $x * y$, $Seg(x,i,j)$, $MSP(x,j)$, $LSP(x,j)$ are computable in A_0.

Definition 2.9. $Maxindex(f,x) = \mu i \leq |x| \, \forall k \leq |x| \, (f(k) \leq f(i))$.

Lemma 2.10. *If $f \in A_0$, then $Maxindex(f,x)$ is in A_0.*

Proof. Although this is a consequence from Proposition 9.4 in [8], we give a direct proof here. For $0 \leq i, j \leq |x|$, define $P(i,j) = 1$ if $f(i) \geq f(j)$, else 0. Now for any $i \leq |x|$ we use CRN to encode a function $F(i,x) = P(i,0) * P(i,1) * \cdots * P(i,|x|)$. Then $f(i)$ is a maximum iff $F(i,x) = 2^{|x|+1} - 1$. Now we may use $\mu i \leq |x| \, [F(i,x) = 2^{|x|+1} - 1]$ to obtain such i. □

Definition 2.11. F is definable from g, h_0, h_1 by k-BRN (*k-bounded recursion on notation*) for $k \in \mathbb{N}$ if

$$\begin{aligned} F(0, \overrightarrow{x}) &= g(\overrightarrow{x}), \\ F(2n, \overrightarrow{x}) &= h_0(n, \overrightarrow{x}, F(n, \overrightarrow{x})) \text{ if } n > 0, \\ F(2n+1, \overrightarrow{x}) &= h_1(n, \overrightarrow{x}, F(n, \overrightarrow{x})), \end{aligned}$$

and $0 \leq F(n, \overrightarrow{x}) \leq k$ for all $n, \overrightarrow{x}$.

Definition 2.12. The function f is defined from g, h_0, h_1, by weak k-BRN (*weak, k-bounded recursion on notation*) if $f(x, \overrightarrow{u}) = F(|x|, \overrightarrow{u})$ and $F(x, \overrightarrow{u})$ is definable from g, h_0, h_1 by k-BRN.

Note that the number of steps in iterated recursions of k-BRN and weak k-BRN are $|x|$ and $||x||$ respectively.

Lemma 2.13. *A_0 is closed under weak k-BRN.*

Proof. See [7] and next remark. □

Remark 2.14. Since k is a constant, we may modify the proof in [7] by assuming that $k+1$ is of the form 2^{2^m}. With this the encoding and decoding of sequences would be much more easier. (Weak multiplication $|x| \cdot |y|$ is not needed.)

Definition 2.15. $count(x)$ is the number of 1's in the binary expression of x, i.e.,

$$\begin{aligned} count(0) &= 0, \\ count(s_0(x)) &= count(x), \text{ provided } x > 0, \\ count(s_1(x)) &= count(x) + 1. \end{aligned}$$

And T_0 is the smallest class containing basic functions $zero$, s_0, s_1, i^n_k, $|x|$, $\#$, $Bit(i,x)$, $count$, and closed under composition and CRN. (That is, $T_0 = A_0 + count$.)

Definition 2.16. *Sharply bounded counting* $sbcount(x, y)$ is defined as follows:

$$sbcount(x,y) = \begin{cases} count(x) & \text{if } x \leq |y|, \\ 0 & \text{else.} \end{cases}$$

This means $count(x)$ for small x.

Lemma 2.17. $sbcount(x, y) \in A_0$.

Proof. The idea is to use sharply bounded μ-operator to compute every bit of $sbcount(x, y)$, and then concatenate those required bits into $sbcount(x, y)$. For detail see Lemma "$BSUM$ is in FO" in [3]. □

Remark 2.18. In [3], the proof of $BSUM \in FO$ has a gap. We fix it as follows. For $x > |y|$ or $x = 0$, we simply output 0. Suppose that $0 < x \leq |y|$. We may assume that $|x|$ is of the form 2^m by the following modification.

Let $\hat{x} = x - 2^{|x|-1} + 2^{2^{|x|_2 - 1}}$. $\hat{x}$ is computable in A_0 and $|\hat{x}| = 2^{|x|_2}$. Since $|y| \geq 1$,

$$4|y|^2 < |y|^6 + |y|^4 + |y|^3 + |y|^2 + |y| + 1 = |\overbrace{y\#y\# \cdots \#y}^{6 \text{ times}}|.$$

Hence

$$\hat{x} \leq 2^{2^{|x|_2}} \leq 2^{2^{|y|_3}} \leq 2^{2|y|_2} \leq (2|y|)^2 \leq |\overbrace{y\#y\# \cdots \#y}^{6 \text{ times}}|.$$

We may use $\hat{y} = \overbrace{y\#y\# \cdots \#y}^{6 \text{ times}}$ instead of y. Therefore $|\hat{x}| = 2^{|x|_2}$, $\hat{x} \leq |\hat{y}|$, and $sbcount(x, y) = sbcount(\hat{x}, \hat{y})$.

The rest of this proof is basically a direct translation from the proof in [3]. Note that CRN, weak k-BRN and sharply bounded μ-operator are used.

Why do we need this modification? Without it, we will need to do weak multiple addition to compute the indices of those bits which we are going to concatenate together (to get $sbcount(x, y)$). As we will see in Section 5, this may need to use *sbcount* again, with smaller size (shrinking by log). Repeating this argument, the applications of *sbcount* or weak multiple addition are not constantly many times! To avoid this, we may need some neat coding tricks. However, the computation of those indices does not need weak multiple addition nor *sbcount* if $|x|$ is of the form 2^m.

Convention. Consider any A_0 function $f(\overrightarrow{x})$. If $f(\overrightarrow{x}) \leq |g(\overrightarrow{x})|$ for some term $g(\overrightarrow{x})$, then $count(f(\overrightarrow{x})) = sbcount(f(\overrightarrow{x}), g(\overrightarrow{x}))$ is in A_0. Hereafter we simply denote $sbcount(f(\overrightarrow{x}), g(\overrightarrow{x}))$ by $sbcount(f(\overrightarrow{x}))$ for sharply bounded $f(\overrightarrow{x})$, and we call this the sharply bounded counting of $f(\overrightarrow{x})$.

Remark 2.19. The resolution of this gap in [3] was pointed out by Carlos Parra, who suggested that "It suffices to consider the case $|x| = 2^m$." With other examples the author realized that this simple assumption is extremely useful. (Also see Lemma 8.9.)

Definition 2.20. For $f \in T_0$, we define $deg_{count}[f]$ the *count* degree of f. $F = \langle f_1, f_2, \ldots, f_m \rangle$ is a *construction sequence* if each f_i is either a basic function in T_0, or a composition of f_j, f_k with $j, k < i$, or f_i is obtained by CRN with $g = f_j, h_0 = f_{k_0}, h_1 = f_{k_1}$, $j, k_0, k_1 < i$. Here we associate the construction sequence F with the way of the construction: encode this by a sequence of m elements, each element is of the forms $\langle 0, i\rangle$ (the i-th basic function), $\langle 1, j, k\rangle$ (composition $f_j \circ f_k$), $\langle 2, j, k_0, k_1\rangle$ (CRN from f_j, f_{k_0}, f_{k_1}). For a construction sequence F, we define $deg_{count}[f_i, F]$ for $i \leq m$ inductively (according to the code of construction):

(1) If f_i is a basic function other than *count*, then $deg_{count}[f_i, F] = 0$.
(2) If $f_i = count$, then $deg_{count}[f_i, F] = 1$.
(3) If $f_i = f_j \circ f_k$ in F, then $deg_{count}[f_i, F] = deg_{count}[f_j, F] + deg_{count}[f_k, F]$.
(4) If f_i is obtained by CRN with g, h_0, h_1 in F, then

$$deg_{count}[f_i, F] = max(deg_{count}[g, F], deg_{count}[h_0, F], deg_{count}[h_1, F]).$$

$deg_{count}[f] = min\{deg_{count}[f, F] : f \in F\}$. $A_0(count)_n \stackrel{def}{=} \{f \in T_0 : deg_{count}[f] \leq n\}$. Such f is called *count* n times sequentially.

3. Bit matrix for multiplication and multiple addition

In this section, we formalize the following: Given n numbers $z(1), z(2), \ldots, z(n)$, we may define a bit function $F : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ by $F(i, j) = Bit(j, z(i))$. If $z(i)$ is an A_0 function, then $F(i, j)$ is obviously in A_0.

Definition 3.1. Function F is called a *bit function* if $Im(F) \subseteq \{0, 1\}$.
Assume that $F : \mathbb{N}^{m+2} \to \{0, 1\}$ and $\overrightarrow{x} = x_1, x_2, \ldots, x_m$, then $F(\overrightarrow{x}, i, j)$ is called *bit matrix* with respect to $\overrightarrow{x}$. Here i, j are indices of row, column. The multiple addition of bit matrix F is defined as follows.

$$Sum(F, \overrightarrow{x}) \stackrel{def}{=} \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} 2^j F(\overrightarrow{x}, i, j).$$

Example 3.2. In elementary school the following method is taught as the first step of multiplication: Let $|x| = n_1$, $|y| = n_2$. First, multiply $Bit(0, y)$ with x, write it down on row 0 from bit $n_1 - 1$ to bit 0, that is, shift to left by one bit. Second, multiply $Bit(1, y)$ with x, write it down on row 1 from bit n_1 to bit 1. (Then shift to left by one bit.) Repeat this write-shift procedure until $Bit(n_2 - 1, y) \cdot x$ is written on row $n_2 - 1$.

Formally,

$$F_0(x, y, i, j) \stackrel{def}{=} \begin{cases} Bit(i, y) \cdot Bit(j - i, x) & \text{if } j \geq i, \\ 0 & \text{else.} \end{cases}$$

Then $F_0 \in A_0$ and $x \cdot y = Sum(F_0, x, y)$.

Example 3.3. Let $z(\overrightarrow{x}, i) \in A_0$ (or T_0). To compute $\sum_{i=0}^{|s(\overrightarrow{x})|} z(\overrightarrow{x}, i)$, the first step is to define bit matrix

$$F_0(\overrightarrow{x}, i, j) = \begin{cases} Bit(j, z(\overrightarrow{x}, i)) & \text{if } i \leq |s(\overrightarrow{x})|, \\ 0 & \text{else.} \end{cases}$$

Then $F_0 \in A_0$ (or T_0) and $Sum(F_0, \overrightarrow{x}) = \sum_{i=0}^{|s(\overrightarrow{x})|} z(\overrightarrow{x}, i)$.

4. Column counting process

In this section, we formalize column counting process (CCP), which is an efficient way to accelerate multiple addition.

Definition 4.1. Bit matrix $F(\overrightarrow{x}, \cdot, \cdot)$ is *column bounded* by $s(\overrightarrow{x})$ if for any $i \geq s(\overrightarrow{x})$ and any j, $F(\overrightarrow{x}, i, j) = 0$. F is *row bounded* by $t(\overrightarrow{x})$ if for any $j \geq t(\overrightarrow{x})$ and any i, $F(\overrightarrow{x}, i, j) = 0$.

Suppose that bit matrix $F_0(\overrightarrow{x}, \cdot, \cdot)$ is column bounded by $s(\overrightarrow{x})$ and row bounded by $t(\overrightarrow{x})$. First apply *count* to column j by $\sum_{i=0}^{s(\overrightarrow{x})-1} F_0(\overrightarrow{x}, i, j)$ for $0 \leq j < t(\overrightarrow{x})$. This can be done in T_0 while $s(\overrightarrow{x})$ is sharply bounded. We may denote column counting for column j by C_j. Next we can generate a new bit matrix F_1 (as in Section 3): write down C_0 at row 0, shift to left by one bit, write down C_1 at row 1, shift to left by one bit, etc. Note that $Sum(F_1, \overrightarrow{x}) = Sum(F_0, \overrightarrow{x})$. However, to sum F_1 is easier because to compute $Sum(F_1, \overrightarrow{x})$ is the same as to compute a multiple addition with $|s(\overrightarrow{x})|$ many numbers, much less than $s(\overrightarrow{x})$ many.

Since *count* may apply to any binary string with polynomial size, inside T_0 we shall only consider the case that F_0 is column bounded by a sharply bounded function.

The following is just the formalization of column counting process. (Those who are not interested in details may skip to Remark 4.14.)

Lemma 4.2. $|x \cdot y| \leq |max(x\#x, y\#y)|$.

Proof. Since $2|x| \leq |x|^2 + 1 = |x\#x|$, this implies

$$|x \cdot y| \leq |x| + |y| \leq max(2|x|, 2|y|) \leq max(|x\#x|, |y\#y|) = |max(x\#x, y\#y)|.$$

□

Example 4.3. Consider $x \cdot y$ and F_0 in Example 3.2. From Lemma 4.2, F_0 is row bounded by $|t(x, y)|$ where $t(x, y) = max(x\#x, y\#y) \in A_0$. Also F_0 is column bounded by $|y|$.

Example 4.4. Consider $\sum_{i=0}^{|s(\overrightarrow{x})|} z(\overrightarrow{x}, i)$ and F_0 in Example 3.3. In this case F_0 is column bounded by $|s(\overrightarrow{x})| + 1$. Since

$$\begin{aligned} |\sum_{i=0}^{|s(\overrightarrow{x})|} z(\overrightarrow{x}, i)| &\leq |(1 + |s(\overrightarrow{x})|) \cdot \max\nolimits_{0 \leq i \leq |s(\overrightarrow{x})|} z(\overrightarrow{x}, i)| \\ &= |(1 + |s(\overrightarrow{x})|) \cdot z(\overrightarrow{x}, Maxindex(z(\overrightarrow{x}, \cdot), s(\overrightarrow{x})))|, \end{aligned}$$

by Lemma 4.2 F_0 is row bounded by $|max(s_1\#s_1, s_2\#s_2)|$. Here $s_1 = 1 + |s(\overrightarrow{x})| \in A_0$ and $s_2 = z(\overrightarrow{x}, Maxindex(z(\overrightarrow{x}, \cdot), s(\overrightarrow{x}))) \in A_0$. Note that $s_2 \in A_0$ is from Lemma 2.10.

Definition 4.5. Bit matrix F is called *upper triangular* if for any $j < i$ and any $\overrightarrow{x}$, $F(\overrightarrow{x}, i, j) = 0$. F is *upper triangular with width* $\leq w(\overrightarrow{x})$ if $F(\overrightarrow{x}, i, j) = 0$ for $j < i$ or $j \geq i + w(\overrightarrow{x})$.

Example 4.6. In Example 3.2, F_0 is upper triangular with width $\leq |x|$.

Lemma 4.7. *If bit matrix $F(\overrightarrow{x}, i, j) \in T_0$ and F is column bounded by $|s(\overrightarrow{x})|$, then $\sum_{i=0}^{\infty} F(\overrightarrow{x}, i, j) \in T_0$.*

Proof. In order to do the column counting, we first encode column $F(\overrightarrow{x}, \cdot, j)$ to number $G(\overrightarrow{x}, j)$. By CRN,

$$G(\overrightarrow{x}, j) \stackrel{def}{=} \sum_{i=0}^{\infty} 2^i F(\overrightarrow{x}, i, j) = \sum_{i=0}^{|s(\overrightarrow{x})|-1} 2^i F(\overrightarrow{x}, i, j) < 2^{|s(\overrightarrow{x})|} < \infty$$

is in T_0. Then $count(G(\overrightarrow{x}, j)) = \sum_{i=0}^{\infty} F(\overrightarrow{x}, i, j) \in T_0$. □

Lemma 4.8. *If bit matrix $F(\overrightarrow{x}, i, j) \in A_0$ and F is column bounded by $||s(\overrightarrow{x})||$, then $\sum_{i=0}^{\infty} F(\overrightarrow{x}, i, j) \in A_0$.*

Proof. Since $G(\overrightarrow{x}, j) < 2^{||s(\overrightarrow{x})||}$ is sharply bounded, $count(G(\overrightarrow{x}, j))$ is in A_0 by sharply bounded counting. □

Definition 4.9. (Column counting process) Let bit matrix $F_0(\overrightarrow{x}, i, j)$ be column bounded by $|s(\overrightarrow{x})|$ and $G(\overrightarrow{x}, j) = \sum_{i=0}^{\infty} 2^i F_0(\overrightarrow{x}, i, j)$, then

$$F_1(\overrightarrow{x}, j, k) \stackrel{def}{=} Bit(k, 2^j count(G(\overrightarrow{x}, j))).$$

We denote this by $CCP(F_0) = F_1$.

Lemma 4.10. *If F_0 is row bounded by $t(\overrightarrow{x})$ and column bounded by $s(\overrightarrow{x})$, and $F_1 = CCP(F_0)$, then $Sum(F_1, \overrightarrow{x}) = Sum(F_0, \overrightarrow{x}) < \infty$.*

Proof. The column bound and row bound guarantee that $Sum(F_0, \overrightarrow{x})$ is finite. The equality can be derived from definition. □

Lemma 4.11. *If $F_0(\overrightarrow{x}, i, j) \in T_0$ is column bounded by $|s(\overrightarrow{x})|$, then $F_1 = CCP(F_0)$ is upper triangular with width $\leq ||s(\overrightarrow{x})||$ and $F_1 \in T_0$.*

Proof. By Definition 4.9 and Lemma 4.7. □

Lemma 4.12. *If $F_1(\overrightarrow{x}, i, j) \in A_0$ is upper triangular with width $\leq ||s(\overrightarrow{x})||$, then $F_2 = CCP(F_1)$ is upper triangular with width $\leq |s(\overrightarrow{x})|_3$ and $F_2 \in A_0$.*

Proof. Similar to Lemma 4.11. Any *count* in this case is in A_0 because of sharply bound counting. □

Remark 4.13. Note that there is no column bound condition in Lemma 4.12. In this case the column counting process is still possible; we may convert F_1 into a column bounded function (see Lemma 5.1).

Remark 4.14. **(Weakness of Column Counting Process)** Suppose that we want to compute multiple addition $\sum_{i=1}^{n} z_i$ by CCP and $|z_i| \leq m$ for $1 \leq i \leq n$. First we define $F_0(i, j) = Bit(j, z_i)$, $F_1 = CCP(F_0)$, $F_2 = CCP(F_1)$, etc. Then the width of f_1 is $|n|$, the width of F_2 is $|n|_2$, etc. Although the sequence $a_l \stackrel{def}{=} |n|_l$ decreases very fast at the beginning, when $a_l = 2$, $a_{l+1} = |2| = 2$. Then $a_{l+2} = a_{l+3} = \cdots = 2$, the sequence stops decreasing, i.e., CCP becomes very inefficient when $a_l = 2$. On the other hand, we are not sure about the existence of T_0 function $H(k,i,j)$ which equals $F_k(i,j)$. (This circuit is mentioned in [5] and [4].) To limit the use of CCP to constant times, we shall give an A_0 algorithm which uses partial sums to compute multiple addition. (See Section 5.)

5. Partial sum algorithm for multiple addition

Suppose that F_0 is column bounded by $s(\overrightarrow{x})$, applying CCP we get $F_1 = CCP(F_0), F_2 = CCP(F_1), \ldots, F_k = CCP(F_{k-1}), \ldots$ such that F_k is upper triangular with width $\leq |s(\overrightarrow{x})|_k$. Lemma 5.1 converts F_k to multiple addition for at most $|s(\overrightarrow{x})|_k$ many numbers.

Lemma 5.1. *If bit matrix $F_k(\overrightarrow{x},i,j)$ is upper triangular with width $\leq |s(\overrightarrow{x})|_k$, then*

$$B(\overrightarrow{x},i,j) \overset{def}{=} \begin{cases} F_k(\overrightarrow{x},i+l,j) & \text{if } j \geq |s(\overrightarrow{x})|_k \text{ and } l = j - |s(\overrightarrow{x})|_k + 1 \\ F_k(\overrightarrow{x},i,j) & \text{else} \end{cases}$$

is column bounded by $|s(\overrightarrow{x})|_k$ and $Sum(B,\overrightarrow{x}) = Sum(F_k,\overrightarrow{x})$.

Proof. B is obtained by shifting upward the nonzero part of each column in F_k so that it is column bounded by $|s(\overrightarrow{x})|_k$. And then the sum of B is the same as the sum of F_k. ☐

Now we consider bit matrix $B(i,j), 1 \leq i \leq s, 0 \leq j \leq t$. (For convenience we always assume that $s > 2$.) Define the multiple addition of B as follows:

$$Sum(B) \overset{def}{=} \sum_{i=1}^{s} \sum_{j=0}^{t} 2^j B(i,j).$$

We may assume that $|Sum(B)| \leq t+1$. (Suppose not, let $t' = t + |s| + 1$, then $|Sum(B)| < |s \cdot 2^{t+1}| = |s| + t + 2 = t' + 1$. That is, we may consider every number x as $0 \ldots 0x$ such that the multiple addition will not overflow.)

Definition 5.2. (Partial sum)
Let s,t fixed, $t \geq l_2 > l_1 \geq 0$,

$$PSum(B,l_2,l_1) \overset{def}{=} \sum_{i=1}^{s} \sum_{j=l_1}^{l_2} 2^j B(i,j)$$

is called the *partial sum* of B between column l_2, l_1.

Remark 5.3. $Sum(B)$ and $PSum(B,l_2,l_1)$ look as follows:

$$\begin{array}{cccccccc}
 & B(1,t) & \cdots & B(1,l_2) & \cdots & B(1,l_1) & \cdots & B(1,0) \\
 & B(2,t) & \cdots & B(2,l_2) & \cdots & B(2,l_1) & \cdots & B(2,0) \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
+) & B(s,t) & \cdots & B(s,l_2) & \cdots & B(s,l_1) & \cdots & B(s,0) \\
\hline
\multicolumn{8}{c}{Sum(B)}
\end{array}$$

$$\begin{array}{rccccccccc} & 0 & \cdots & 0 & B(1,l_2) & \cdots & B(1,l_1) & 0 & \cdots & 0 \\ & 0 & \cdots & 0 & B(2,l_2) & \cdots & B(2,l_1) & 0 & \cdots & 0 \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ +) & 0 & \cdots & 0 & B(s,l_2) & \cdots & B(s,l_1) & 0 & \cdots & 0 \\ \hline & & & & & PSum(B,l_2,l_1) & & & & \end{array}$$

Lemma 5.4. *If $l_3 > l_2 > l_1$, then*

$$Bit(l_2, PSum(B, l_3, l_1)) = Bit(l_2, PSum(B, l_2, l_1)).$$

Proof. $PSum(B, l_3, l_1) = PSum(B, l_3, l_2+1) + PSum(B, l_2, l_1)$. Since the first par is a multiple of 2^{l_1+1}, $Bit(l_2, PSum(B, l_3, l_1)) = Bit(l_2, PSum(B, l_2, l_1))$. □

The key idea is that if $l_2 - l_1 \geq |s|+1$, then $Bit(l_2, Sum(B))$ is almost determined.

Lemma 5.5. *If $l_2 - l_1 \geq |s|+1, t \geq l_2 > l_1 \geq 0$, and $Bit(l_2 - 1, PSum(B, l_2, l_1)) = 0$, then $Bit(l_2, Sum(B)) = Bit(l_2, PSum(B, l_2, l_1))$.*

Proof. By Lemma 5.4, $Bit(l_2, Sum(B)) = Bit(l_2, PSum(B, l_2, 0))$. It suffices to consider $PSum(B, l_2, 0)$. Now $PSum(B, l_2, 0) = PSum(B, l_2, l_1) + PSum(B, l_1 - 1, 0)$. The tail part $PSum(B, l_1 - 1, 0) \leq s \cdot (2^{l_1} - 1) < 2^{l_1+|s|}$. Since $l_2 - 1 \geq l_1 + |s|$, adding the tail part to $PSum(B, l_2, l_1)$ at most changes the $(l_2 - 1)$-th bit from 0 to 1. This implies $Bit(l_2, Sum(B)) = Bit(l_2, PSum(B, l_2, l_1))$. □

Remark 5.6. Actually Lemma 5.5 shows that "carry may occur at most once." Since $PSum(B, l_1 - 1, 0)$ is quite small, if we add it to $PSum(B, l_2, l_1)$ bit-by-bit, during this procedure the $(l_2 - 1)$-th bit may alter at most once. That is, the status of the $(l_2 - 1)$-th bit could be: $0 \to 0 \to \cdots \to 0$ (always 0), $1 \to 1 \to \cdots \to 1$ (always 1), $0 \to \cdots \to 0 \to 1 \to \cdots \to 1$ (from 0 to 1), $1 \to \cdots \to 1 \to 0 \to \cdots \to 0$ (from 1 to 0). And the subsequences $0 \to 1 \to 0$, $1 \to 0 \to 1$ are not possible.

For convenience we identify $PSum(B, l, l') = PSum(B, l, 0)$ for $l' < 0$.

In Lemma 5.5, if $Bit(l_2 - 1, PSum(B, l_2, l_1)) = 1$, then we may compare two partial sums to see whether the carry occurs.

Theorem 5.7. *Let $l_2 - l_1 \geq |s| + 1$.*

(1) *If $Bit(l_2 - 1, PSum(B, l_2, l_1)) = Bit(l_2 - 1, PSum(B, l_2 - 1, l_1 - 1)) = 1$, then $Bit(l_2, PSum(B, l_2, l_1)) = Bit(l_2, PSum(B, l_2, l_1 - 1))$.*
(2) *If $Bit(l_2 - 1, PSum(B, l_2, l_1)) = 1$ and $Bit(l_2 - 1, PSum(B, l_2 - 1, l_1 - 1)) = 0$, then $Bit(l_2, PSum(B)) = 1 - Bit(l_2, PSum(B, l_2, l_1))$.*

Proof. By Lemma 5.4, $Bit(l_2 - 1, PSum(B, l_2, l_1 - 1)) = Bit(l_2 - 1, PSum(B, l_2 - 1, l_1 - 1))$. Now $PSum(B, l_2, l_1 - 1) = PSum(B, l_2, l_1) + 2^{l_1 - 1} \sum_{i=1}^{s} B(i, l_1 - 1)$. When we add the tail part $2^{l_1 - 1} \sum_{i=1}^{s} B(i, l_1 - 1)$ to $PSum(B, l_2, l_1)$, according to Remark 5.6, two cases may happen:

(1) The status of the $(l_2 - 1)$-th bit is $1 \to 1 \to \cdots \to 1$ (always 1): In this case $Bit(l_2, PSum(B, l_2, l_1)) = Bit(l_2, PSum(B, l_2, l_1 - 1))$.
(2) The status of the $(l_2 - 1)$-th bit is $1 \to \cdots \to 1 \to 0 \to \cdots \to 0$: In this case $Bit(l_2, PSum(B, l_2, l_1 - 1)) = 1 - Bit(l_2, PSum(B, l_2, l_1))$. Since $Bit(l_2 - 1, PSum(B, l_2, l_1 - 1)) = 0$, by Lemma 5.5

$$Bit(l_2, Sum(B)) = 1 - Bit(l_2, PSum(B, l_2, l_1)).$$

□

Now we define $PS_k(B,l) = PSum(B,l,l-k)$, and we will just consider the case $k \geq |s|+1$. Then by induction we have the following generalization of Theorem 5.7.

Theorem 5.8. *Let $k \geq |s|+1$, and define $a(l) = Bit(l, PS_k(B,l)), b(l-1) = Bit(l-1, PS_k(B,l))$, which look as follows:*

$$\begin{array}{lclllllll} PS_k(B,l) & : & \cdots & a(l) & b(l-1) & \cdots & \cdots & \cdots \\ PS_k(B,l-1) & : & \cdots & \cdots & a(l-1) & b(l-2) & \cdots & \cdots \\ PS_k(B,l-2) & : & \cdots & \cdots & \cdots & a(l-2) & b(l-3) & \cdots \\ \vdots & & \cdots & \cdots & \cdots & \cdots & \ddots & \ddots \end{array}$$

(1) *If $b(l-1) = b(l-2) = \cdots = b(m) = 1$, and $a(l-1) = a(l-2) = \cdots = a(m) = 1$, then $Bit(l, PSum(B,l,m)) = Bit(l, PS_k(B,l))$.*

(2) *If $b(l-1) = b(l-2) = \cdots = b(m) = 1$, $a(l-1) = b(l-2) = \cdots = a(m+1) = 1$, and $a(m) = 0$, then $Bit(l, Sum(B)) = 1 - Bit(l, PS_k(B,l))$.*

By Theorem 5.8, Remark 5.6, we have:

Algorithm 5.9. (Partial sum algorithm)

Let $a(l) = Bit(l, PS_k(B,l)), b(l-1) = Bit(l-1, PS_k(B,l))$ for all l, and let $k \geq |s|+1$, then the function $Bit(l, Sum(B))$ is determined by the following algorithm:

(1) If $b(l-1) = 0$, then $Bit(l, Sum(B)) = a(l)$.

(2) If $b(l-1) = b(l-2) = \cdots = b(m) = 1, b(m-1) = 0$, then:

  (a) If there exist an h such that $l > h \geq m$, then $Bit(l, Sum(B)) = 1 - a(l)$.

  (b) Else $Bit(l, Sum(B)) = a(l)$.

Remark 5.10. If $a(l), b(l)$ are given, then the determination of $Bit(l, Sum(B))$ is computable in A_0

6. Partial sum in A_0

In this section, we show how to compute partial sum in A_0 when the width of upper triangular bit matrix is quite small.

Lemma 6.1. *For any $c \geq 1$, $c|x|_2$ is double sharply bounded.*

Proof. $c|x|_2 \leq ||2x|^c| \leq |\overbrace{(2x)\#(2x)\#\cdots\#(2x)}^{c \text{ times}}|_2$. □

Lemma 6.2. *$|x|_3 \cdot |y|_3$ is double sharply bounded.*

Proof. Let $t_c(x) \overset{def}{=} \overbrace{(2x)\#(2x)\#\cdots\#(2x)}^{c \text{ times}}$.

If $x = 1$ or $y = 1$, the proof is obvious. We may assume that $x, y \geq 2$. Note that $x \geq 2$ implies $|x|_n \geq 2$ for all n. If $\sqrt{|x|_2} \geq |x|_3$ and $\sqrt{|y|_2} \geq |y|_3$, then the following inequality holds:

$$\begin{aligned} |x\#y|_2 &\geq ||x| \cdot |y|| \\ &\geq |x|_2 + |y|_2 - 1 \\ &\geq 2\sqrt{|x|_2|y|_2} - 1 \\ &\geq 2|x|_3 \cdot |y|_3 - 1 \\ &\geq |x|_3|y|_3. \end{aligned}$$

Note that $A = \{x : \sqrt{|x|_2} < |x|_3\}$ is a finite set. Let c be an upper bound of $\{|x|_3 : \sqrt{|x|_2} < |x|_3\}$. By Lemma 6.1, for $x \in A$, we have

$$|x|_3 \cdot |y|_3 \leq c|y|_3 \leq |t_c(|y|)|_2.$$

(So does y.) Hence

$$|max(t_c(|x|), t_c(|y|), x\#y)|_2 = max(|t_c(|x|)|_2, |t_c(|y|)|_2, |x\#y|_2) \geq |x|_3 \cdot |y|_3.$$

□

Lemma 6.3. $2^{|x|_4} \cdot |y|_3$ *is double sharply bounded.*

Proof. By Lemmas 6.2 and 6.1,

$$2^{|x|_4} \cdot |y|_3 \leq 2|x|_3 \cdot |y|_3 \leq 2|t|_2 \leq |(2t)\#(2t)|_2$$

for some t. □

Example 6.4. Fix $k \geq 3$. If an A_0 function $z(\overrightarrow{x}, i) < |u(\overrightarrow{x})|_k (< 2^{|u|_{k+1}})$ for $0 \leq i < |v(\overrightarrow{x})|_k$, then the following A_0 algorithm computes $\sum_{i=0}^{|v(\overrightarrow{x},i)|_k - 1} z(\overrightarrow{x}, i)$:

(1) $y(\overrightarrow{x}, i) = 2^{z(\overrightarrow{x},i)} - 1$. Such y is double sharply bounded. So it is computable in A_0 by sharply bounded search.

(2) Concatenate $y(\overrightarrow{x}, 0), y(\overrightarrow{x}, 1), \ldots, y(\overrightarrow{x}, |v(\overrightarrow{x})|_k)$ into a binary sequence by the following way:

$$G(\overrightarrow{x}, j) = \begin{cases} Bit(l, y(\overrightarrow{x}, w)) & \text{if } j = w \cdot 2^{|u|_{k+1}} + l, \\ & \quad 0 \leq l < 2^{|u|_{k+1}}, \\ & \quad 0 \leq w < |v(\overrightarrow{x})|_k; \\ 0 & \text{else.} \end{cases}$$

The definition of $G(\overrightarrow{x}, j)$ is obviously in A_0.

(3) Now define $H(\overrightarrow{x}) = \sum_{j=0}^{|v|_k \cdot 2^{|u|_{k+1}}} 2^j \cdot G(\overrightarrow{x}, j)$. $H(\overrightarrow{x}) \in A_0$ is by CRN. Note that $H(\overrightarrow{x})$ is sharply bounded by Lemma 6.3. Now sharply bounded counting implies that

$$sbcount(H(\overrightarrow{x})) = \sum_{i=0}^{|v(\overrightarrow{x},i)|_k - 1} z(\overrightarrow{x}, i)$$

is computable in A_0.

Remark 6.5. In Example 6.4, we avoid using weak multiplication $|x| \cdot |y|$, which may cause circular argument.

Theorem 6.6. *Multiplication* $x \cdot y$ *is in* T_0. *Furthermore, there is an algorithm which computes* $x \cdot y$ *and uses count once sequentially.*

Proof. From Example 3.2, we get the bit matrix for $x \cdot y$. Now we apply CCP twice so that the width is about $|y|_3$. By Lemma 5.1, we can convert the bit function into the column bounded form which we may apply multiple addition. From Example 6.4, we know how to calculate partial sum in A_0. By partial sum algorithm, $Bit(i, x \cdot y)$ can be determined. Note that only at the first CCP we do need the function *count*. In the rest of this computation sharply bounded counting (in A_0) will be sufficient. □

Corollary 6.7. *Multiplication* $x \cdot |y|$ *is in* A_0.

Proof. Similar to Theorem 6.6, except that sharply bounded counting is sufficient for the first CCP. □

Remark 6.8. Now we apply partial sum algorithm to numerical approximation. Suppose that $u = \sum_{i=1}^{\infty} v_i$, and $1 > v_1 \geq v_2 \geq \cdots \geq 0$. (The upper bound 1 is for simplicity.) We can express real number v_i by $\sum_{j=1}^{\infty} F(i,j) \cdot 2^{-j}$, where $Im(F) \subseteq \{0,1\}$. Then the multiple addition of F is u. Column j is called bounded by $p(j)$ if for all $i > p(j), F(i,j) = 0$. Now we focus on cases with polynomial column bounds, say, $p(j) = cj^k$ with constant $c > 0$ and integer $k > 0$. We have the following similar result: To determine $Bit(-t,u)$, it suffices to consider the partial sum

$$PS(F;t) = \sum_{j=t}^{s(t)} \sum_{i=0}^{p(s(t))} F(i,j)2^{-j},$$

where $s(t) = t + (k+\epsilon)|t| + 2 + d$, $\epsilon > 0$, and constant d depends on c and ϵ. Note that the number 2 provides the first two bits $a(\cdot), b(\cdot)$ in Algorithm 5.9. Let

$$\begin{aligned} a(-t) &= Bit(-t, PS(F;t)), \\ b(-t-1) &= Bit(-t-1, PS(F;t)). \end{aligned}$$

Now we may extend the partial sum algorithm over negative integers to determine $Bit(-t,u)$. (This may not be in A_0 for it may not halt.)

7. Sharply bounded multiple product

In this section we show that multiple products with sharply bounded values are computable in A_0. First we show that exponentiation with sharply bounded value is computable in A_0 (Theorem 7.6). Then multiple products with sharply bounded values can be factorized as $\prod_{i=1}^{k} p_i^{a_i}$ in A_0. With this one can use sharply bounded search to find out the least positive integer which is a multiple of $p_i^{a_i}$ for $i = 1$ to k. (Note that k,$p_i^{a_i}$ are all sharply bounded, and then $p_i^{a_i}$ is computable in A_0.)

Lemma 7.1. *The function*

$$dsbrem(x,y,u) = \begin{cases} x - \lfloor x/y \rfloor \cdot y & \text{if } 0 < y \leq |u|_2 \\ 0 & \text{else} \end{cases}$$

is in A_0.

Proof. Since the rational number $1/y$ has periodic binary expression, and its period has length $< y \leq |u|_2$, we can use sharply bounded μ-operator to compute its periodic part and non-periodic part in A_0. With this it is easy to compute quotient and remainder. □

$x(\text{mod} y)$ is defined as $\mu i \leq y\,[y \mid (x-i)]$. If x, y are both sharply bounded, then $\lfloor x/y \rfloor, x(\text{mod} y)$ are computable in A_0.

Lemma 7.2. *The function*

$$h_1(x,s,y,u) = \begin{cases} x^{2^s}(\text{mod} y) & \text{if } 0 < y \leq |u|_2, s \leq |u|_3 \\ 0 & \text{else} \end{cases}$$

is in A_0.

Proof. Since $x^{2^s}(\text{mod} y) < y \leq |u|_2$, its size is bounded by $|u|_3$. We may construct a short sequence $\langle a_0, a_1, \ldots, a_s\rangle$ such that $a_i = x^{2^i}(\text{mod} y)$. Then $a_{i+1} = {a_i}^2(\text{mod}\ y)$ and it is computable in A_0.

Now the size of this short sequence $\leq |u|_3 \cdot |u|_3$ is double sharply bounded by Lemma 6.2. Hence by sharply bounded μ-operator this sequence (and then a_s) is computable in A_0. ☐

Lemma 7.3. *The function*

$$h_2(x, t, y, u) = \begin{cases} x^t(\text{mod} y) & \textit{if } 0 < y \leq |u|_2, t \leq |u|_2 \\ 0 & \textit{else} \end{cases}$$

is in A_0.

Proof. Let $t = \sum_{i=0}^{s} t_i \cdot 2^i, t_i \in \{0, 1\}$, then $x^t(\text{mod} y) = \prod_{i:t_i=1} a_i(\text{mod} y)$ where a_i is defined in Lemma 7.2. Now define

$$b_{i+1} = \begin{cases} b_i \cdot a_{i+1}(\text{mod} y) & \text{if } t_{i+1} = 1, \\ b_i & \text{else.} \end{cases}$$

By sharply bounded μ-operator the short sequence $\langle b_0, \ldots, b_s\rangle$ and $x^t(\text{mod} y)$ are computable in A_0. ☐

Lemma 7.4. *The predicate*

$$P(p, u) = \begin{cases} 1 & \textit{if } p \textit{ is a prime and } p \leq |u| \\ 0 & \textit{else} \end{cases}$$

is in A_0.

Let p_i be the i-th prime number. (Question: For sharply bounded p_i, is the function $i \mapsto p_i$ in A_0? It is *count* once in T_0.)

Lemma 7.5. *If t is large enough, then* $2^t < \prod_{p_i \leq t} p_i$.

Proof. By prime number theorem, when t is large enough, $1 - \epsilon < [\log_e \prod_{p_i \leq t} p_i]/t$ for small $\epsilon > 0$. Then

$$2^t < 2^{[\log_e \prod_{p_i \leq t} p_i]/(1-\epsilon)} = (2^{(1/1-\epsilon)})^{\log_e \prod_{p_i \leq t} p_i} < \prod_{p_i \leq t} p_i.$$

(We can choose ϵ such that $2^{(1/1-\epsilon)} < e$.) ☐

Theorem 7.6. *The sharply bounded exponentiation function*

$$E(x, t, u) = \begin{cases} x^t & \textit{if } x^t \leq |u| \\ 0 & \textit{else} \end{cases}$$

is in A_0.

Proof. We may assume that $x > 1, t > 1$. If $x^y \leq |u|$, then it is easy to show that $y \leq |u|_2$. Hence $x^t = \mu i \leq |u|\, [i = x^t]$. By Lemma 7.5, we can replace "$i = x^t$" by "$i \equiv x^t(\text{mod} y)$ for $y \leq |u|_2$." Lemmas 7.1,7.3, and sharply bounded quantification show that it is computable in A_0. ☐

Theorem 7.7. *If z, t are in A_0 and $t(x)$ is double sharply bounded, then the sharply bounded multiple product function*

$$f(x,u) = \begin{cases} \prod_{j=1}^{t(x)} z(x,j) & \text{if the product} \leq |u| \\ 0 & \text{else} \end{cases}$$

is in A_0.

Proof. First, each $z(x,j) \leq |u|$. Hence we can factorize $z(x,j)$ to $\prod_{p:P(p,u)} p^{r(x,j,p)}$ in A_0, where

$r(x,j,p) = \mu k \leq |u| \, \exists w, v \leq |u| \, [P(p,u) \wedge p^k = w \wedge w \cdot v = z(x,j) \wedge v(\text{mod} p) \neq 0]$.

(This is computable in A_0 by Lemma 7.4, Theorem 7.6, and the fact that "sharply bounded multiplication and division are computable in A_0.") Then

$$\prod_{j=1}^{t(x)} z(x,j) = \prod_{p \leq |u|} p^{\sum_{j=1}^{t(x)} r(x,j,p)}.$$

Since $t(x)$ is double sharply bounded, $\sum_{j=1}^{t(x)} r(x,j,p)$ is computable in A_0.

Now $c_p = p^{\sum_{j=1}^{t(x)} r(x,j,p)} \leq |u|$ is computable in A_0 by Theorem 7.6. Instead of constructing a bit matrix for multiple addition (as in Section 8) we use sharply bounded μ-operator:

$$\prod_{j=1}^{t(x)} z(x,j) = \mu w \leq |u| \, [w > 0 \wedge \forall p < |u| \, [P(p,u) \rightarrow c_p \mid w]].$$

Note that w is sharply bounded and hence $c_i \mid w$ can be verified in A_0. □

Now if $t(x)$ is sharply bounded in Theorem 7.7, will the sharply bounded multiple product be in A_0? If $\prod_{j=1}^{t(x)} z(x,j) \leq |u|$, then $|\{j \leq t(x) : z(x,j) > 1\} \leq |u|_2$. The crucial part will be: Is the sparse counting function (with respect to polynomial q)

$$spcount(x) = \begin{cases} count(x) & \text{if } count(x) \leq q(|x|_2) \\ 0 & \text{else} \end{cases}$$

computable in A_0? The answer is yes.

Lemma 7.8. *$spcount(x) \in A_0$.*

Proof. We use the idea in [14], Lemma 3: there are about $n/log_e n$ many primes less than $n (= |x|)$. If $count(x) \leq |n|^k$ (k is the degree of q), then there exists a small prime p such that $\forall i, j \leq n \, [i \neq j, Bit(i,x) = Bit(j,x) = 1 \rightarrow i \not\equiv j (\text{mod} p)]$.

Now we prove this claim. Consider $\Delta = \{i - j : j < i \leq n, Bit(i,x) = Bit(j,x) = 1\}$, then $|\Delta| < |n|^{2k}$. Each $a \in \Delta$ has less than $|n|$ many prime factors. Then in the first $|n|^{2k+1}$ primes there is a prime p such that $p \nmid a$ for all $a \in \Delta$. When n is large enough, $p \leq |n|^{4k+2}$. We can first compute p by sharply bounded μ-operator. Then we construct a binary string with length $\leq p \leq |n|^{4k+2}$ and it has the same cardinality as $\{i \leq n : Bit(i,x) = 1\}$: for $0 \leq j \leq p-1$, $b(j) = 1$

iff $\exists i \leq n\,[i \equiv j(\mathrm{mod} p)]$. (This maps $\{i \leq n : Bit(i,x) = 1\}$ into $\{j : j < p\}$ injectively.) Now by CRN, Lemma 2.17, $|\{j < p : b(j) = 1\}|$ is computable in A_0. □

Theorem 7.9. *If z, t are in A_0 and $t(x)$ is sharply bounded, then the sharply bounded product function*

$$f(x,u) = \begin{cases} \prod_{j=1}^{t(x)} z(x,j) & \text{if the product} \leq |u| \\ 0 & \text{else} \end{cases}$$

is in A_0.

Proof. Similar to Theorem 7.7. First we use Lemma 7.8 to verify that $|\{j \leq t(x) : z(x,j) > 1\}| \leq |u|_2$. For the sum $\sum_{j=1}^{t(x)} r(x,j,p)$ we can use Lemma 7.8 and partial sum algorithm. Hence it is computable in A_0. □

Remark 7.10. Theorem 7.6 is inspired by [15], in which "$c^{|x|_2} \in$ uniform AC^0 for constant c" is proved. That proof uses Lemma 7.5 and Nepomnjaščij's technique. In this paper the corresponding function algebraic part of Nepomnjaščij's technique is probably the sharply bounded μ-operator and the fact "$|x|_3 \cdot |y|_3$ is double sharply bounded."

8. Multiple product with polylogarithmic size

We are going to explore the power of partial sum algorithm a little bit more. Here we investigate exponentiation and multiple product. Although we know that 2^x is too large for NC to compute, some kind of weak exponentiation is computable in T_0.

If bit function f is column bounded by s and row bounded by t, f can be seen as a matrix of size $\leq s \times t$. We may imagine that number x as a $1 \times |x|$ matrix. If we do not use CCP and partial sum algorithm, the bit matrix for product $x \cdot y$ has size $\leq |y| \times (|x| + |y|)$. Then if we multiply the bit matrix of $x \cdot y$ and the bit matrix of z, we will get a bit matrix with size $\leq (|y| \cdot |z|) \times (|x| + |y| + |z|)$.

Example 8.1. Define F_1 for $x \cdot y$ as follows:

$$F_1(x,y,i,j) = \begin{cases} Bit(i,y) \cdot Bit(j-i,x) & \text{if } j \geq i, \\ 0 & \text{else.} \end{cases}$$

Then the size of $F_1 \leq |y| \times (|x| + |y|)$ and $Sum(F_1, x, y) = x \cdot y$:

$$\begin{aligned} Sum(F_1,x,y) &= \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} 2^j F_1(x,y,i,j) \\ &= \sum_{i=0}^{\infty}\sum_{j\geq i} 2^j Bit(i,y) \cdot Bit(j-i,x) \\ &\text{(Set } u = j - i.\text{)} \\ &= \sum_{i=0}^{\infty} 2^i Bit(i,y) [\sum_{u=0}^{\infty} 2^u Bit(u,x)] \\ &= y \cdot x. \end{aligned}$$

Now we use F_1 and z to define F_2 for $(x \cdot y) \cdot z$:

$$F_2(x,y,z,k,l) = \begin{cases} Bit(j,z) \cdot F_1(x,y,i,l-j) & \text{if } l \geq j, \\ & \quad 0 \leq i < |y|, \\ & \quad 0 \leq j < |z|, \\ & \quad k = |y| \cdot j + i; \\ 0 & \text{else.} \end{cases}$$

It is easy to verify that the size of $F_2 \leq (|y| \cdot |z|) \times (|x| + |y| + |z|)$.

Lemma 8.2. *In Example 8.1, $F_2 \in A_0$ and $Sum(F_2, x, y, z) = x \cdot y \cdot z$.*

Proof. Since the size of $F_2 \leq (|y| \cdot |z|) \times (|x| + |y| + |z|)$, $F_2(x,y,z,k,l) = 0$ for $k > |y| \cdot |z|$ or $l > |x| + |y| + |z|$. Hence we only need to consider sharply bounded k, l. The multiplication $|y| \cdot j$ and the sharply bounded search of j are in A_0 because j is sharply bounded. Hence $F_2 \in A_0$. The equality is easy to verify. □

With F_2, CCP, and partial sum algorithm, the computation for $x \cdot y \cdot z$ only needs to use *count* once sequentially.

Remark 8.3. Given function $f(\overrightarrow{x})$, if there is an A_0 bit matrix $F(\overrightarrow{x}, i, j)$, which is column sharply bounded, and $Sum(F, \overrightarrow{x}) = f(\overrightarrow{x})$, then $f \in A_0(count)_1$.

In Example 8.4 we make uniform the size of numbers x, y, z to 2^m. With the same trick we may compute $F_n(\overrightarrow{x}, i, j)$ by some A_0 function $F(n, \overrightarrow{x}, i, j)$. (See Lemma 8.9.)

Example 8.4. Let $m = max(||x||, ||y||, ||z||)$, then $2^m > max(|x|, |y|, |z|)$. Define F_1 for $x \cdot y$:

$$F_1(x,y,i,j) = \begin{cases} Bit(i,y) \cdot Bit(j-i,x) & \text{if } j \geq i, \\ & \quad i < 2^m; \\ 0 & \text{else.} \end{cases}$$

Then $F_1 \in A_0$, the size of $F_1 \leq 2^m \times (2^m + 2^m)$, and $Sum(F_1, x, y) = x \cdot y$.

Now we define F_2 from F_1, z for $x \cdot y \cdot z$:

$$F_2(x,y,z,k,l) = \begin{cases} Bit(j,z) \cdot F_1(x,y,i,l-j) & \text{if } l \geq j, \\ & \quad 0 \leq i < 2^m, \\ & \quad 0 \leq j < 2^m, \\ & \quad k = j \cdot 2^m + i; \\ 0 & \text{else.} \end{cases}$$

Then $F_2 \in A_0$. This is much easier to compute because MSP takes over the job which is previously done by weak multiplication. It is obvious that $Sum(F_1, x, y) = x \cdot y$ and $Sum(F_2, x, y, z) = x \cdot y \cdot z$. We get this advantage by allowing more 0's in the bit matrix.

Consider $\prod_{i=0}^{n} z(i)$. Suppose that $2^m > max_{0 \leq i \leq n}(|z(i)|)$. We may define bit matrices $F_1, F_2, \ldots, F_n$ as we have done in Example 8.4. By induction, F_n is of size $\leq 2^{mn} \times (n+1)2^m$. In order to do CCP, we need to force 2^{mn} to be sharply bounded. Lemma 8.5 shows that it is possible for some special cases.

Lemma 8.5. *If $n \leq |y|_2/|y|_3$ and p is a polynomial, then $p(||x||)^n$ is sharply bounded.*

Proof. Since $p(||x||)^n \leq 2^{k\cdot|x|_3\cdot n}$ for some constant k, it suffices to show that $|x|_3 \cdot n$ is double sharply bounded.

We prove this in 4 cases:

(1) If $|y|_2 = 1$ (or 2), then $|y|_2/|y|_3 = n = 1$. Hence $|x|_3 \cdot n$ is double sharply bounded.
For the following three cases we may assume that $|y|_3 \geq 2$.

(2) If $x \leq y$, then $|x|_3 \cdot n \leq |x|_3 \cdot |y|_2/|y|_3 \leq |y|_2$.

(3) If $|y|_3 \leq |x|_3 \leq 3|y|_3$, then
$$|x|_3 \cdot n \leq |x|_3 \cdot |y|_2/|y|_3 \leq 3|y|_2 \leq ||2y|^3| \leq |(2y)\#(2y)\#(2y)|_2.$$

(4) If $m|y|_3 \leq |x|_3 \leq (m+1)|y|_3$ for some $m \geq 3$: Let $b = |y|_3, a = |x|_3$, this implies $2^{b-1} \leq |y|_2 \leq 2^b - 1$. Then $|x|_3 \cdot |y|_2/|y|_3 \leq a(2^b-1)/b$.
Consider double sharply bounded term $|x\#y|_2$:
$$|x\#y|_2 = |2^{|x||y|}|_2 \geq ||x|\cdot|y|| \geq |x|_2 + |y|_2 - 2 \geq 2^{a-1} + 2^{b-1} - 2.$$
(This is from fact $|p\cdot q| \geq |p| + |q| - 2$.) It suffices to show that $a(2^b-1)/b \leq 2^{a-1} + 2^{b-1} - 2$.
From the hypothesis $(m+1)b > a \geq mb$, $a(2^b-1)/b \leq (m+1)(2^b-1)$. We need to show that $(m+1)(2^b-1) \leq 2^{mb-1} + 2^{b-1} - 2 (\leq 2^{a-1} + 2^{b-1} - 2)$ for $m \geq 3$ and $b \geq 2$. This can be easily proved by induction on m, b.

□

Remark 8.6. In general, if $n \leq |y|_2/|y|_{k+1}$ for $k \geq 2$, then $p(|x|_k)^n$ is sharply bounded.

Remark 8.7. Original idea of Lemma 8.5: For $y < x$, if $|y|_2/|y|_3$ is monotone increasing, then
$$|x|_3 \cdot |y|_2/|y|_3 \leq |x|_3 \cdot |x|_2/|x|_3 \leq |x|_2$$
is double sharply bounded. Unfortunately $|y|_2/|y|_3$ is not monotone increasing. However, it doesn't damage Lemma 8.5.

Example 8.8. Given $z(\overrightarrow{x}, i), m = m(\overrightarrow{x}) = max_{0\leq i\leq n}(||z(\overrightarrow{x}, i)||)$. We may define bit matrix $F(n, \overrightarrow{x}, \cdot, \cdot)$ for $\prod_{i=0}^{n} z(\overrightarrow{x}, i)$ as follows:
$$F(1, \overrightarrow{x}, i, j) = \begin{cases} Bit(i, z(\overrightarrow{x}, 1)) \cdot Bit(j-i, z(\overrightarrow{x}, 0)) & \text{if } j \geq i, \\ & \quad i < 2^m; \\ 0 & \text{else.} \end{cases}$$
For all $n \geq 1$, define
$$F(n+1, \overrightarrow{x}, k, l) = \begin{cases} Bit(j, z(\overrightarrow{x}, n+1)) \cdot F(n, \overrightarrow{x}, i, l-j) & \text{if } l \geq j, \\ & \quad 0 \leq i < 2^{nm}, \\ & \quad 0 \leq j < 2^m, \\ & \quad k = j \cdot 2^{nm} + i; \\ 0 & \text{else.} \end{cases}$$
It is easy to verify that $Sum(F(n, \cdot, \cdot, \cdot), \overrightarrow{x}) = \prod_{i=0}^{n} z(\overrightarrow{x}, i)$ by the technique in Lemma 8.2 and induction.

Lemma 8.9. *In Example 8.8, if $n \leq |t(\overrightarrow{x})|$ and $z(\overrightarrow{x}, i) \in T_0$, then $F(n, \overrightarrow{x}, k, l) \in T_0$. If $n \leq |t(\overrightarrow{x})|_2$ and $z(\overrightarrow{x}, i) \in A_0$, then $F(n, \overrightarrow{x}, k, l) \in A_0$.*

Proof. Consider $n \leq |t(\overrightarrow{x})|$. $F(n, \overrightarrow{x})$ is of size $\leq 2^{mn} \times (n+1)2^m$, where 2^{mn} is computable in A_0. For any $k < 2^{mn}$, let

$$k = j_n \cdot 2^{m(n-1)} + j_{n-1} \cdot 2^{m(n-2)} + \cdots + j_1$$

with $0 \leq j_w < 2^m$ for $1 \leq w \leq n$. Inductively

$$F(n, \overrightarrow{x}, k, l) = \begin{cases} \prod_{w=2}^{n} Bit(j_w, z(\overrightarrow{x}, w)) \cdot F(1, \overrightarrow{x}, j_1, l - \sum_{w=2}^{n} j_w) & \text{if } l \geq \sum_{w=2}^{n} j_w, \\ 0 & \text{else.} \end{cases}$$

Since $j_w (\leq 2^m)$ and n are all sharply bounded, $\sum_{w=2}^{n} j_w$ is in T_0. If n is double sharply bounded and $z \in A_0$, then $\sum_{w=2}^{n} j_w$ is in A_0.

□

Remark 8.10. The 2^m size assumption is the key which makes simple recursion possible in Lemma 8.9. In this case it depends on multiple addition $\sum_{w=2}^{n} j_w$.

Theorem 8.11. *If $z(\overrightarrow{x}, i) \in A_0$, p is a polynomial, $n \leq |y|_2/|y|_3$, $m = |\overrightarrow{x}|$, and $|z(\overrightarrow{x}, i)| \leq p(\log m)$ for $0 \leq i \leq n$, then multiple product $Prod(z; \overrightarrow{x}, y, n) = \prod_{i=0}^{n} z(\overrightarrow{x}, i)$ is count once in T_0.*

Proof. From Lemma 8.9, F for $\prod_{i=0}^{n} z(\overrightarrow{x}, i)$ is definable in A_0. Lemma 8.5 shows that $F(n, \overrightarrow{x}, \cdot, \cdot)$ is column bounded by sharply bounded terms. Hence CCP is applicable. Now we can apply partial sum algorithm to F for $\prod_{i=0}^{n} z(\overrightarrow{x}, i)$. □

Since the product in Theorem 8.11 still has polylogarithmic size, we can apply Theorem 8.11 iteratively to get:

Theorem 8.12. *If $z(\overrightarrow{x}, i) \in A_0$, p is a polynomial, $n \leq (|y|_2/|y|_3)^k$, $m = |\overrightarrow{x}|$, and $|z(\overrightarrow{x}, i)| \leq p(\log m)$ for $0 \leq i \leq n$, then multiple product $Prod(z; \overrightarrow{x}, y, n) = \prod_{i=0}^{n} z(\overrightarrow{x}, i) \in A_0(count)_k$.*

Corollary 8.13. *If $z(\overrightarrow{x}, i) \in A_0$, p, q are polynomials, $deg(q) = k$, $n \leq q(||y||)$, $m = |\overrightarrow{x}|$, and $|z(\overrightarrow{x}, i)| \leq p(\log m)$ for $0 \leq i \leq n$, then $Prod(z; \overrightarrow{x}, y, n) = \prod_{i=0}^{n} z(\overrightarrow{x}, i)$ is computable in $A_0(count)_{k+1}$.*

Proof. It is because $q(||y||) \ll (|y|_2/|y|_3)^{k+1}$. □

Actually the restriction "$n \leq q(||y||)$" is not necessary: if $|\prod_{i=0}^{n} z(\overrightarrow{x}, i)| \leq q(\log m)$ and $n \leq |\overrightarrow{x}|$, then $|\{i : z(\overrightarrow{x}, i) > 1\}| \leq q(\log m)$. By Lemma 7.8, we can define another A_0 function $\check{z}$ and $l \leq q(\log m)$ such that $\check{z}(\overrightarrow{x}, i) > 1$ for $i \leq l$ and $\prod_{i=0}^{l} \check{z}(\overrightarrow{x}, i) = \prod_{i=0}^{n} z(\overrightarrow{x}, i)$. Hence we have:

Corollary 8.14. *If $z(\overrightarrow{x}, i) \in A_0$, p is a polynomial of degree k, $n \leq m = |\overrightarrow{x}|$, and $|\prod_{i=0}^{n} z(\overrightarrow{x}, i)| \leq q(\log m)$, then $\prod_{i=0}^{n} z(\overrightarrow{x}, i)$ is computable in $A_0(count)_{k+1}$.*

We shall give some examples. A trivial one is $|x|^{p(|y|_2)}$: it is computable in $A_0(count)_{deg(p)+1}$.

Example 8.15. $n!$ for $n \leq |x|_2/|x|_3$ is computable in A_0, for it is sharply bounded. To compute $(|x|_2)!$, it suffices to divide this product into $|x|_3$ many subproducts:

$$(|x|_2)! = \left(\prod_{i=1}^{\frac{|x|_2}{|x|_3}} i\right)\left(\prod_{i=\frac{|x|_2}{|x|_3}+1}^{2\frac{|x|_2}{|x|_3}} i\right)\cdots\left(\prod_{i=(|x|_3-1)\frac{|x|_2}{|x|_3}+1}^{|x|_2} i\right).$$

(Minor adjustment to round up numbers like $|x|_2/|x|_3$ to integers is needed.) Now each subproduct is sharply bounded and computable in A_0. (Each subproduct $\leq (|x|_2)^{|x|_2/|x|_3} \leq 2^{|x|_3 \cdot |x|_2/|x|_3} = 2^{|x|_2}$.) Now the product of subproducts is computable in $A_0(count)_1$ by Theorem 8.11 since $|x|_3 \leq |x|_2/|x|_3$ for all but finite x.

Example 8.16. By Stirling's formula, $\binom{|y|_2}{i}$ is sharply bounded. Hence $\binom{|y|_2}{i} \in A_0$.

Now we consider the case that $z(\overrightarrow{x}, i)$ is not sharply bounded, but with a simple form.

Example 8.17. Consider $(2^{|x|}+1)^{|y|_2}$. We define the corresponding F as follows:

$$F(1,x,i,j) = \begin{cases} Bit(j, 2^{|x|}+1) & \text{if } i = 0; \\ Bit(j-|x|, 2^{|x|}+1) & \text{if } i = 1, \\ & \quad j \geq |x|; \\ 0 & \text{else.} \end{cases}$$

and for $n \geq 1$,

$$F(n+1,x,k,l) = \begin{cases} F(n,x,k,l) & \text{if } k < 2^n; \\ F(n,x,i,l-|x|) & \text{if } l \geq |x|, \\ & \quad k = 2^n + i, \\ & \quad 0 \leq i < 2^n; \\ 0 & \text{else.} \end{cases}$$

Inductively,

$$F(n,x,k,l) = \begin{cases} F(1,x,Bit(0,k), l - count(\lfloor k/2 \rfloor)) & \text{if } l \geq count(\lfloor k/2 \rfloor), \\ & \quad k < 2^{n-1}; \\ 0 & \text{else.} \end{cases}$$

If $n \leq |y|_2$, then $F(n,\cdot,\cdot,\cdot)$ is computable in A_0 (because $\lfloor k/2 \rfloor \leq 2^{n-2}$ is sharply bounded). Since F is column sharply bounded, we can use CCP and partial sum algorithm to compute $(2^{|x|}+1)^{|y|_2}$ in $A_0(count)_1$. By this, we can also compute $\binom{|y|_2}{i} \in A_0(count)_1$:

$$(2^{|x|}+1)^{|y|_2} = \sum_{i=0}^{|y|_2} \binom{|y|_2}{i} 2^{i \cdot |x|}.$$

Since $\binom{|y|_2}{i}$ is sharply bounded, we may choose $x = y$ so that

$$\binom{|y|_2}{i} = Seg((2^{|y|}+1)^{|y|_2}, i \cdot |y|, (i+1)|y| - 1).$$

However, multiple addition does worse than sharply bounded search in this example: we shall compute $\binom{|y|_2}{i}$ in A_0 and use $\sum_{i=0}^{|y|_2} \binom{|y|_2}{i} 2^{i \cdot |x|}$ for $(2^{|x|}+1)^{|y|_2}$.

Example 8.18. Consider

$$(\sum_{i=0}^{|x|-1} 2^i)^{|y|_2}.$$

This is computable in A_0 by $(2^{|x|}-1)^{|y|_2}$. If we consider the multiple addition for this, we may construct F with range $\{0,1,-1\}$; $\sum_{i=0}^{|x|-1} 2^i = 2^{|x|}-1 = 1\cdot 2^{|x|}+(-1)\cdot 2^0$, which is a sequence with elements $0,1,-1$. As in Example 8.17, F is column sharply bounded. For the counting we can separate F into two parts, positive part P and negative part N:

$$\begin{aligned} P(n,x,y,i,j) &= \begin{cases} F(n,x,y,i,j) & \text{if } F(n,x,y,i,j) \geq 0, \\ 0 & \text{else.} \end{cases} \\ N(n,x,y,i,j) &= \begin{cases} -F(n,x,y,i,j) & \text{if } F(n,x,y,i,j) \leq 0, \\ 0 & \text{else.} \end{cases} \end{aligned}$$

By summing up both parts $Sum(P), Sum(N)$, we get $(2^{|x|}-1)^{|y|_2} = Sum(P) - Sum(N)$.

However, we may apply the trick in Example 8.18 to show the following result.

Lemma 8.19. *If $z_1(\overrightarrow{x}, i), z_2(\overrightarrow{x}, i) \in A_0$, $n \leq |y|_2/|y|_3$, p is a polynomial, $m = |\overrightarrow{x}|$, and $count(z_1(\overrightarrow{x}, i)), count(z_2(\overrightarrow{x}, i)) \leq p(\log m)$ for $i \leq n$, then $\prod_{i=1}^{n}(z_1(\overrightarrow{x}, i) - z_2(\overrightarrow{x}, i))$ is computable in $A_0(count)_1$.*

Examples 8.16,8.18 show some weakness of the multiple addition method. Now we show more of that.

Remark 8.20. Suppose that we want to compute $(|x|_2)!$ by defining F in a very economic way. First we define a T_0 function h which searches the position of the i-th non-zero bit:

$$h(i,x) = \begin{cases} j & \text{if } \sum_{k=0}^{j} Bit(k,x) = i, \\ 0 & \text{else.} \end{cases}$$

Then h is computable in T_0 by sharply bounded search. (Question: is it computable in A_0 when i is $p(||x||)$ for some polynomial p? The answer is yes.)

Now we define S_F for the size of F:

$$S_F(1) = 1, S_F(n+1) = S_F(n) \cdot count(n+1).$$

Then F is defined as follows:

$$\begin{aligned} F(1,x,i,j) &= \begin{cases} Bit(j,1) & \text{if } i = 0; \\ 0 & \text{else.} \end{cases} \\ F(n+1,x,k,l) &= \begin{cases} F(n,x,j,l-h(i,n+1)) & \text{if } k = S_F(n)\cdot(i-1)+j, \\ & \quad 1 \leq i \leq count(n+1), \\ & \quad 0 \leq j < S_F(n), \\ & \quad l \geq h(i,n+1); \\ 0 & \text{else.} \end{cases} \end{aligned}$$

The intuitive idea of F is from Example 8.17: When we multiply the bit matrix with a new number $(n+1)$, we simply skip those zero bits in the binary expression of $(n+1)$. Hence the column size of $F(n,\cdot,\cdot,\cdot)$ is $S_F(n) = \prod_{i=1}^{n} count(i)$, and for

$r \le S_F(n)$, the r-th row has non-zero entry. (Actually every row has exactly one non-zero entry, but this is not important—any product defined in Example 8.8 with $count(z(\overrightarrow{x}, 0)) = 1$ has this property.)

We do not know whether $F(|x|_2, x, \cdot, \cdot)$ is computable in T_0. Anyway, even $F(|x|_2, x, \cdot, \cdot)$ is computable in T_0, it doesn't help because the column size is not sharply bounded!

Lemma 8.21. $\prod_{n=1}^{2^m-1} count(n) = \prod_{i=1}^{m} i^{\binom{m}{i}}$. *If* $m = |x|_3$, *then* $\prod_{n=1}^{2^m-1} count(n)$ *is not sharply bounded.*

Proof. The first part is by induction. From induction hypothesis

$$\prod_{n=2^m}^{2^{m+1}-1} count(n) = \prod_{n=0}^{2^m-1} (1 + count(n)) = \prod_{i=1}^{m} (i+1)^{\binom{m}{i}} = \prod_{i=2}^{m+1} i^{\binom{m}{i-1}}.$$

Then

$$\begin{aligned}\prod_{n=1}^{2^{m+1}-1} count(n) &= \prod_{n=1}^{2^m-1} count(n) \cdot \prod_{n=2^m}^{2^{m+1}-1} count(n) = \prod_{i=1}^{m} i^{\binom{m}{i}} \cdot \prod_{i=2}^{m+1} i^{\binom{m}{i-1}} \\ &= \prod_{i=2}^{m} i^{\binom{m}{i}+\binom{m}{i-1}} \cdot (m+1) = \prod_{i=2}^{m} i^{\binom{m+1}{i}} \cdot (m+1) = \prod_{i=1}^{m+1} i^{\binom{m+1}{i}}.\end{aligned}$$

Now we estimate $\prod_{n=1}^{2^m-1} count(n)$. For $1 \le k < m$, $k(m-k) \ge m-1$. Then

$$k^{\binom{m}{k}} (m-k)^{\binom{m}{m-k}} = [k(m-k)]^{\binom{m}{k}} \ge (m-1)^{\binom{m}{k}}.$$

Applying this for all $k \in [1, m-1]$, we get

$$\prod_{i=1}^{m} i^{\binom{m}{i}} \ge (m-1)^{\frac{2^m-2}{2}} m.$$

Then

$$(m-1)^{\frac{2^m-2}{2}} \ge (|x|_3/2)^{\frac{\frac{|x|_2}{2}-2}{2}} \ge 2^{(|x|_4-2)\cdot|x|_2/8} \ge |x|^{(|x|_4-2)/8}.$$

□

Remark 8.22. We may combine the techniques in Example 8.18, Remark 8.20 to see whether it is possible to compute $|x|_2!$. We construct the bit matrix in the following way:

(1) Express every number by $\sum b_i 2^i$ with $b_i \in \{-1, 0, 1\}$ such that $|\{i : b_i \neq 0\}|$ is minimal in all possible expressions. (See Example 8.18.)
(2) Omit all those 0 rows. (See Remark 8.20.)

Although we can construct the bit matrix in such an economic way, the matrix size is not sharply bounded: First we denote by $alt(x)$ the alternation of x. (For example, $x = (110011101)_2$, $11 \to 00 \to 111 \to 0 \to 1$, then $alt(x) = 5$.) Then $\prod_{n=1}^{2^m-1} alt(n) = \prod_{i=1}^{m} i^{\binom{m}{i}}$. (The proof is similar to Lemma 8.21.) Since the size of minimal expression for n is $\Theta(alt(n))$, the size of the corresponding matrix for $|x|_2!$ is not sharply bounded.

Remark 8.23. The remaining hope we may have for multiple addition method is that if each column of the bit matrix is sparse, then we can apply Lemma 7.8 and compute it in A_0. This seems unlikely. We may consider the computation of $[(|x|_2/|x|_3)!]^2$ with $|x|_2/|x|_3 = 2^m - 1$ for some m. First $[(|x|_2/|x|_3)!]^2$ is sharply bounded, then by Theorem 7.9 it is computable in A_0. Now we show that there is a column with more than $(\log n)^k$ many 1's for any constant k.

Consider the F constructed for $[(|x|_2/|x|_3)!] \cdot [(|x|_2/|x|_3)!]$ as in Example 8.8. Consider the m bits binary expression of $i = b^i_m b^i_{m-1} \dots b^i_2 b^i_1 < 2^m - 1$ in the first $[(|x|_2/|x|_3)!]$, then there is a $rev(i) = b^i_1 b^i_2 \dots b^i_{m-1} b^i_m$ in the second $[(|x|_2/|x|_3)!]$. If we choose bit $b^i_k = 1$ from i in the first product, we then automatically choose bit $b^i_k = 1$ from $rev(i)$ in the second product so that $b^i_k \cdot 2^{k-1} \cdot b^i_k \cdot 2^{m-k} = 2^{m-1}$. There are $2^m - 1$ many i's in the first product. Hence any product of this kind (for every i, choose $b^i_k = 1$ from i at the first part and then automatically do the same choice from $rev(i)$) will always have the value $2^{(m-1)\cdot(2^m-1)}$, that is, it appears at column $(m-1)\cdot(2^m-1)$. (Note that unit bit is at column 0.) Then how many 1's of this kind will appear in column $(m-1)\cdot(2^m-1)$? $\prod_{i=1}^{2^m-1} count(i)$.

Now by Lemma 8.21 we estimate a lower bound for $\prod_{i=1}^{2^m-1} count(i)$:

$$\prod_{i=1}^{2^m-1} count(i) \geq (|x|_3/2)^{\frac{|x|_2/|x|_3-1}{2}} \geq 2^{(|x|_4-2)\cdot\frac{|x|_2/|x|_3-1}{2}}.$$

For any constant k, $(\log n)^k \leq 2^{k|x|_3}$. It is quite obvious that

$$(|x|_4 - 2)\cdot\frac{|x|_2/|x|_3 - 1}{2} \gg k|x|_3.$$

Concluding Remark: Now we see enough evidences of the weakness of multiple addition method. However it seems to be the only method (as far as I know) to compute some not-sharply-bounded multiple products inside weak uniform TC^0. We now summarize the possible steps (according to this paper) to minimize the use of *count* in multiple products:

(1) Partition the product into subproducts with sharply bounded values, then compute subproducts in A_0 (by Theorem 7.9): this may eliminate some unnecessary use of multiple addition method.
(2) If some numbers have sizes $\geq p(\log n)$ for any polynomial p, we shall expect that there are no more than $O(\log n/\log\log n)$ many, and all but finitely many of them can be expressed as $z_1 - z_2$ by sparse z_1, z_2: in this case we can apply Lemma 8.19. (If there are more than $(\log n/\log\log n)^k$ many numbers with non-polylogarithmic size for $k > 1$, we will have very little chance to compute the result by Lemma 8.19: we need to have all but finite of them expressible as $z_1 - z_2$ by sparse z_1, z_2, and then there is a partition of subproducts (for these numbers) such that all but finite subproducts have sparse difference expression. And we will need to apply Lemma 8.19 repeatedly, at each time the above condition holds, untile we get the final result.)

Research problems:

(1) Although $x^{|y|}$ is known in P-uniform TC^0, it is not known to be in L. A simpler question is: $3^{|x|} \in T_0$? And a more simpler one is: $3^{||x||^2} \in A_0(count)_1$?

(2) Is $A_0(count)_k$ for $k > 0$ strictly increasing? This problem is not the same as $TC_1^0 \subsetneq TC_2^0 \subsetneq TC_3^0$ in [10], for we allow using AC^0 circuit inside. A natural attempt for this separation problem is "$3^{||x||^k} \in A_0(count)_k \backslash A_0(count)_{k-1}$?"

The final goal of problem (2) is to separate TC^0 and NC^1 by showing that $A_0(count)_k$ does not collapse. (On the other hand, the function $tree(x)$, defined as an OR-AND alternating tree on the input x, needs to be used at most once sequentially for any uniform NC^1 function. This is from the fact "$tree$ is complete in NC^1 under AC^0 or $DLOGTIME$ reduction." See [6]. For a direct function algebraic proof, see [13].)

Acknowledgments. Sections 3,4, 5,6,8 are from the author's Ph.D. thesis [13]. I gratefully acknowledge the many valuable suggestions of Professors Ward Henson, Carl Jockusch, Lou van den Dries, and Kequan Ding during the preparation of the thesis. I also wish to thank Heng-Huat Chan, Peter Clote, Kousha Etessami, Neil Immerman, Jan Krajíček, Carlos Parra, Pavel Pudlák for helpful discussions, suggestions, and comments. Finally, I want to express my deepest gratitude to my advisor Professor Gaisi Takeuti, who initiated this work by a very "simple" question: How to do multiplication in uniform NC^1?

References

[1] M. Ajtai. Σ_1^1 formulae on finite structures. *Annals of Pure and Applied Logic*, 24:1–48, 1983.

[2] E. Allender and V. Gore. A uniform circuit lower bound for the permanent. *SIAM Journal on Computing*, 23(5):1026–1049, October 1994.

[3] D. A. M. Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. In *SCT: Annual Conference on Structure in Complexity Theory*, pages 47–59, 1988.

[4] D. A. M. Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *Journal of Computer and System Sciences*, 41:274–306, 1990.

[5] A. K. Chandra, L. Stockmeyer, and U. Vishkin. Constant depth reducibility. *SIAM Journal on Computing*, 13(2):423–439, 1984.

[6] P. Clote. Sequential machine independent characterizations of the parallel complexity classes AlogTIME, AC^k, NC^k, and NC. In *Feasible Mathematics: A Mathematical Sciences Institute Workshop held in Ithaca, New York, June 1989*, pages 49–69. Birkhäuser, 1990.

[7] P. Clote. On polynomial size frege proofs of certain combinatorial principles. In *Clote & Krajíček (Eds.), Arithmetic, Proof Theory, and Computational Complexity*, pages 162–184. Clarendon Press, 1993.

[8] P. Clote and G. Takeuti. First order bounded arithmetic and small boolean circuit complexity classes. In *Feasible Mathematics II: A Mathematical Sciences Institute Workshop*, pages 154–218. Birkhauser, 1995.

[9] M. Furst, J. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17:13–27, 1984.

[10] A. Hajnal, W. Maass, P. Pudlák, M. Szegedy, and G. Turán. Threshold circuits of bounded depth. *Journal of Computer and System Sciences*, 46(2):129–154, 1993.

[11] J. Håstad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the 18-th Annual ACM Symposium on Theory of Computing*, pages 6–20, 1986.

[12] N. Immerman. Expressibility and parallel complexity. *SIAM Journal of Computing*, 18(3):625–638, June 1989.

[13] J.-L. Lee. *Count and tree in uniform NC^1*. PhD thesis, Department of Mathematics, University of Illinois at Urbana-Champaign, 1997.

[14] J. B. Paris and A. Wilkie. Counting Δ_0 sets. *Fundamenta Mathematicae*, 127:67–76, 1986.

[15] C. M. Parra. *Uniformity and bounded arithmetic below P*. PhD thesis, Department of Mathematics, University of Illinois at Urbana-Champaign, 1996.

[16] A. Razborov. Lower bounds on the size of bounded-depth networks over a complete basis with logical addition. *Mathem. Notes of the Academy of Sci. of the USSR*, 41(4):333–338, 1987.

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

[18] A. C. Yao. Separating the polynomial-time hierarchy by oracles. In *26-th Annual Symposium on Foundations of Computer Science*, pages 1–10, 1985.

Mathematical institute, Academy of Sciences of the Czech Republic
E-mail address: `lee@math.cas.cz`