Csanky's Algorithm

Tushar Mopuri

CCT Presentation

2020

Theorem

Theorem: Let $A = (a_{ij})_{i,j \in [n]}$ be a matrix where all a_{ij}'s are b-bit integers. Then the following can be computed in $(\log nb)^{O(1)}$ time using $(nb)^{O(1)}$ many processors:

1. The characteristic polynomial of A, $p_A(x)$
2. The determinant of A, $det(A)$
3. The inverse of A, A^{-1}

Overview and Assumptions

Essentially, we are trying to show that the problems of computing $p_A(x)$, $det(A)$ and A^{-1} are in NC. References are mentioned at the end, and as instructed the following assumptions are made:

1. Uniformity will not be addressed in this presentation.
2. We assume that integer addition and multiplication are in NC, but will give a more explicit statement in the following slides.

Integer Addition and Multiplication

The proofs for both the following statements are given in Lecture 30 of Dexter C. Kozen - The Design and Analysis of Algorithms (Henceforth called Reference 1). In fact it also gives a method to do integer division with remainder in NC that is based on Newton's method.

- We will use the fact that addition of two n-bit binary numbers can be performed in $\log n$ time with n processors.
- Similarly, we use the fact that two n-bit binary numbers can be multiplied in $O(\log n)$ time with $O(n^2)$ processors. More precisely, the computation is done in $O(\log n)$ stages, each of which takes $O(1)$ time and $O(n^2)$ processors.

Preliminary Properties

Given a matrix $A \in \mathbb{R}^{n \times n}$ (In our case $\mathbb{Z}^{n \times n}$, but since the properties hold more generally, we will state them as such), let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the eigenvalues of A. Then:

- $\text{Det(A)} = \lambda_1 \lambda_2 \ldots \lambda_n$
- $\text{Trace(A)} = \sum_{i=1}^{n}(A_{ii}) = \lambda_1 + \lambda_2 + \cdots + \lambda_n$
- $\text{Trace}(A^i) = (\lambda_1^i + \lambda_2^i + \cdots + \lambda_n^i)$
- **Cayley Hamilton Theorem**: Every square matrix satisfies its characteristic equation. Let $p_A(x) = Det(\lambda I - A) = x^n + c_1 x^{(n-1)} + \cdots + c_n$, then $p_A(A) = A^n + c_1 A^{n-1} + \cdots + c_n I = 0$, where $c_n = (-1)^n Det(A)$.

Computing the product of any two matrices is in NC

WLOG it is enough to consider only $n \times n$ square matrices. Given $X_{n\times n}, Y_{n\times n} \in \mathbb{Z}^{n\times n}$, we can compute $P_{n\times n} = X_{n\times n} \cdot Y_{n\times n}$ as follows:

- Let P_{ij} be the element in the i^{th} row and the j^{th} column of P, then $P_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$. This term can be computed using n integer multiplications and $(n-1)$ integer additions.
- The n multiplications can be done in parallel by n processors requiring $O(1)$ time each, and these outputs can be plugged into another 'stage' which performs the (n-1) additions in a treelike fashion, thus requiring $O(\log n)$ time and $O(n)$ processors for the whole process (These processors can be considered 'arithmetic' processors and we assume they can compute addition and multiplication 'efficiently').

Computing the product of any two matrices is in NC

- Thus computing each P_{ij} is in NC, since the input data A_{ik}, B_{kj} can be assumed to be known values for all $i, j, k \in [n]$. Thus all n^2 P_{ij} values can be computed parallely in $O(\log n)$ time with $O(n^3)$.
- Therefore computing the product of two matrices is in the class NC. (We could call this class NC^A since we are technically using arithmetic processors until we prove that this is equivalent to regular processors, but we assume the proof is already done for notational simplicity).

Computing A^i is in NC

- Given $1 \leq i \leq n$, we claim computing A^i is in NC. We have just shown that finding A^2 is in NC.
- So using repetitive squaring, we can compute any A^i mentioned above using at most $O(\log n)$ matrix multiplications, implying that the problem of computing A^i is in the class NC.

Newton Identities

For $1 \leq i \leq n$, $P_i(\lambda_1, \lambda_2, \ldots, \lambda_n)$ is defined to be the sum of the i^{th} power of the eigenvalues. So,

$$P_1 = \sum_{j=1}^{n} \lambda_j$$

$$P_2 = \sum_{j=1}^{n} \lambda_j^2$$

$$\vdots$$

$$P_n = \sum_{j=1}^{n} \lambda_j^n$$

Newton Identities

Similarly for $1 \leq i \leq n$, $E_i(\lambda_1, \lambda_2, \ldots, \lambda_n)$ is defined to be the sum of all distinct products of i distinct variables. So,

$$E_1 = \sum_{j \in [n]} \lambda_j$$

$$E_2 = \sum_{i<j} \lambda_i \lambda_j$$

$$E_3 = \sum_{i<j<k} \lambda_i \lambda_j \lambda_k$$

$$\vdots$$

$$E_n = \lambda_1 \lambda_2 \ldots \lambda_n$$

Newton Identities

- For all $1 \leq i \leq n$, $P_i = \text{Trace}(A^i)$, and we have already shown that computing A_i is in NC.
- Since the sum contains n terms, the trace can be computed in a treelike manner in $O(\log n)$ time with n processors (Again assuming that addition can be done 'effeciently').
- Therefore from now on, for all $i \in [n]$, P_i can be treated as a known value.

Newton Identities

- The characteristic polynomial $p_A(x) = x^n + c_1 x^{(n-1)} + \cdots + c_n$ is a monic polynomial of degree n, with roots $\lambda_1, \lambda_2, \ldots, \lambda_n$.
- So it can be written as $p_A(x) = (x - \lambda_1)(x - \lambda_2) \ldots (x - \lambda_n)$.
- This implies that $\forall k \in [n]$,

$$c_k = (-1)^k E_k$$

Newton Identities

- From the definitions of P_i and E_i we can say:

$$\begin{aligned} E_1 &= P_1 \\ 2E_2 &= (-)P_2 + P_1E_1 \\ 3E_3 &= (+)P_3 - P_2E_1 + P_1E_2 \\ &\vdots \end{aligned}$$

In general,

$$\begin{aligned} kE_k &= (-1)^{k+1}(P_k - P_{k-1}E_1 + P_{k-2}E_2 - \dots \\ &\quad + (-1)^{k-1}P_1E_{k-1}) \end{aligned}$$

$$\implies E_k = \frac{(-1)^{k+1}(P_k - P_{k-1}E_1 + P_{k-2}E_2 - \dots P_1E_{k-1})}{k}$$

- These relations are called Newton's Identities.

Newton Identities

- We can write these relations in matrix notation as follows:

$$P = MC$$

$$\text{Where, } P = \begin{bmatrix} -P_1 \\ -(P_2)/2 \\ \vdots \\ -(P_{n-1})/(n-1) \\ -P_n/n \end{bmatrix}_{n\times 1} , \; C = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix}_{n\times 1}$$

Newton Identities

- And

$$M = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ -P_1/2 & 1 & 0 & \dots & 0 \\ P_2/3 & -P_1/3 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (-1)^{n-1}P_{n-1}/n & \dots & P_2/n & -P_1/n & 1 \end{bmatrix}_{n \times n}$$

- Then C can be computed by

$$C = M^{-1} \cdot P$$

Newton Identities

- To compute M^{-1} we note that M can be written as the sum of an identity matrix and a nilpotent matrix, as $M = I + N$ where,

$$N = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ -P_1/2 & 0 & 0 & \dots & 0 \\ P_2/3 & -P_1/3 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (-1)^{n-1}P_{n-1}/n & \dots & P_2/n & -P_1/n & 0 \end{bmatrix}_{n \times n}$$

It is easy to see that $N^i = 0$, $\forall i > n$.

Newton Identities

$$\text{And } I = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & 1 \end{bmatrix}_{n \times n}$$

- Hence,

$$\begin{aligned} M^{-1} &= (I + N)^{-1} \\ &= I + N + N^2 + N^3 + \dots \\ &= I + N + N^2 + N^3 + \dots + N^n \\ &\text{(Since } N^i = 0, \forall i > n.\text{)} \end{aligned}$$

Newton Identities

- We have already shown that every N^j for $j \leq n$ can be computed in NC. Thus we can compute M^{-1} parallelly with a polynomially bounded number of processors.
- And now since matrix multiplication is in NC, the entire process of multiplying the output of this stage with P, to obtain $M^{-1} \cdot P$ is in NC.
- With this, we have shown that the computation of the values of all the coeffecients $c_1, c_2, \ldots, c_n$ is in NC, implying the computation of the characteristic polynomial of A is in NC.

Computation of Determinant and Inverse

- As stated earlier, the determinant of the Matrix A, $Det(A) = (-1)^n c_n$, thus it follows directly from the above that this computation is in NC as well.
- Recall from the Cayley Hamilton Theorem that $p_A(A) = A^n + c_1 A^{n-1} + \cdots + c_n I = 0$. If A is non-singular, then $c_n = (-1)^n Det(A) \neq 0$ and A^{-1} exists.
- Then we can multiply this equation by A^{-1} to get:

$$A^{(n-1)} + c_1 A^{(n-2)} + \cdots + c_n A^{-1} = 0$$

$$\implies A^{-1} = \frac{A^{(n-1)} + c_1 A^{(n-2)} + \cdots + c_{n-1} I}{-c_n}$$

Computation of Determinant and Inverse

- We have already shown that the computation of all the coeffecients c_i, $\forall i \in [n]$ is in NC, and since computing A^i is also in NC.
- Thus the computation of A^{-1} is also in NC, since all c_i and A^i, $\forall i \in [n]$ can be computed parallely in NC, and given these outputs, we can compute each of the n^2 elements of A^{-1} parallelly in NC, thus making the whole process NC.

More on 'Effecient' Addition and Multiplication

- Here we will explain what we meant by 'arithmetic processors' being able to 'effeciently' add and multiply integers, and also argue that integer size does not blow-up.
- Clearly, adding or multiplying the b-bit integers given as input can be done in constant time (As b is a constant and nothing in these processes would then depend on n).
- However, the size of the sum of two m-bit integers will be at most $m + 1$ bits (Since we can consider both summands to be of size m, where m is the size of the larger summand).
- Similarly, the size of the product of an m-bit and an $m^{'}$-bit integer will be at most $m + m^{'} + 3$ bits. Both of these observations follow from looking at the maximum value of the product or sum.

More on 'Effecient' Addition and Multiplication

- For the purposes of this presentation, we will consider a rational number a to be represented as a pair of integers, a_1 and a_2, the numerator and the denominator respectively.
- The number of bits required to represent a is at most twice the number of bits required to represent the larger of the two numbers a_1 and a_2.
- For the next few slides, we will use the term 'size' of a rational a, $S(a)$, to denote this number $max(|a_1|, |a_2|)$. So then the number of bits required to represent a is at most twice the 'size' of a.
- The product of two rational a and b, say $c = a \cdot b$ is represented as $(c_1, c_2) = (a_1 \cdot b_1, a_2 \cdot b_2)$. Clearly $S(c)$ is at most $S(a) + S(b) + 3$.

More on 'Effecient' Addition and Multiplication

- Similarly, say $c = a/b$, then $(c_1, c_2) = (a_1 \cdot b_2, a_2 \cdot b_1)$. Clearly $S(c)$ is at most $S(a) + S(b) + 3$.
- The sum of two rational a and b, say $c = a + b$ is represented as $(c_1, c_2) = (a_1 \cdot b_2 + b_1 \cdot a_2, a_2 \cdot b_2)$. Clearly $S(c)$ is at most $S(a) + S(b) + 4$.
- Negation of a rational can by convention be obtained by negating the numerator, and negative rationals can by convention be represented with a negative numerator.
- The computation of the difference of two rationals should follow naturally from this convention and the computation of a sum (A Turing Machine can encode the integers using any enumeration, it will not not affect our results).

More on 'Effecient' Addition and Multiplication

- We will now prove that the 'arithmetic' processors can indeed perform 'effecient computation'. What we meant by this is that each arithmetic processor can be replaced with processors that conform to the definition of processors in effecient parallel computation defined in Lecture 9, and still have the whole computation be in NC (While the output remains unchanged).
- Clearly processors performing arithmetic on input integers of b-bit length each can do so in constant time, with constant memory and need not be replaced.
- As long as we can show that the integers being multiplied or added by any 'arithmetic processors' have size bounded by a polynomial function in nb, say $(nb)^c$, each arithmetic processors can be replaced by at most $O((nb)^{2c})$ processors, taking $O(\log((nb)^c))$ time to give the same output as the 'arithmetic processor'.

More on 'Effecient' Addition and Multiplication

- After replacing all the arithmetic processors, the number of processors has been increased by a factor polynomial in nb, and the time taken to decide has been increased by a factor of $O(\log nb)$.
- Thus a language L that can be decided efficiently in parallel using 'arithmetic processors' (w.r.t n), can also be decided efficiently in parallel using processors that have $O(\log n^2 b)$ bits of memory and perform a poly-time computation at every step (w.r.t. (nb)).
- The most 'taxing' computation an arithmetic processor would have to do in the evaluation of C is $N^n \cdot P$.

More on 'Effecient' Addition and Multiplication

- To obtain the elements of P, we needed to compute A^i, $1 \leq i \leq n$. All elements of A are given to be b-bit integers. An element of $A^2 = B$ is obtained by summing n products of two b-bit integers . This value is at most n times the value of the largest of these products (Like $S(b_{ij}) \leq S(\sum_{k=1}^{n} |a_{ik} \cdot a_{kj}|) \leq S(n \cdot |a_{ik'} \cdot a_{k'j}|)$ for some k'). Thus the size of an element of A^2 is at most $\log n + 2 \cdot b + 6$.
- Now any element of $A^3 = C$ is obtained as $S(c_{ij}) \leq S(\sum_{k=1}^{n} |b_{ik} \cdot a_{kj}|) \leq S(n \cdot |b_{ik'} \cdot a_{k'j}|)$ for some k'. So any element of A^3 is of size at most $\log n + S(b_{ik'}) + S(a_{k'j}) + 6 \leq 2 \cdot \log n + 3 \cdot b + 12$.
- In this manner, an element of A^n has size at most $b + (n-1)(\log n + b + 6)$, which can be easily shown to be polynomial in nb.
- Hence computation of A^i is in NC, with traditional processors as well.

More on 'Effecient' Addition and Multiplication

- The largest elements of P or N involve P_n, the trace of A_n.
- The size of any element of P or N is then at most $n + (nb)^{c'} \leq (nb)^c$. This is because these elements contain sums of n elements of A_n in the numerator, and integers less than n, and thus of size $\leq \log n$, in the denominator.
- The notion of size used in this section is that explained above, which differs by a factor of 2 from the number of bits required.
- By the same argument as above, the largest element of N^n can then be obtained to be of size at most $((nb)^c)^d = (nb)^{cd}$, which is still polynomial in nb.
- To work this out, let $N^2 = M$, then $S(m_{ij}) \leq S(\sum_{k=1}^{n} |S(n_{ik} \cdot n_{kj}|) \leq S(n \cdot |n_{ik'} \cdot n_{k'j}|)$ for some $k' \in [n]$.

More on 'Effecient' Addition and Multiplication

- It is useful to note that it is easy to reduce the rational to a 'lowest terms form' since division is also in NC.
- In the earlier manner, an element of N^n has size at most $(nb)^c + (n-1)(\log n + (nb)^c + 6)$, which can be easily shown to be $\leq (nb)^{d'}$, for some constant d'.
- Thus in the computation of $N^n \cdot P$, n products of integers whose size is polynomial in (nb) are computed, implying that the size of the products are also polynomial, and then added, all of which can be done in NC.
- Hence even the most taxing computation required of an arithmetic circuit have inputs of polynomial size, implying that all 'arithmetic processors' can be replaced by regular processors without losing efficient parallel computation.

Conclusion

Thus we have shown that given $A = (a_{ij})_{i,j \in [n]}$, a matrix where all a_{ij}'s are b-bit integers, the following can be computed in $(\log nb)^{O(1)}$ time using $(nb)^{O(1)}$ many processors:

1. The characteristic polynomial of A, $p_A(x)$
2. The determinant of A, $det(A)$
3. The inverse of A, A^{-1}

References

1. Lectures 30-31 of (Texts and Monographs in Computer Science) Dexter C. Kozen - The Design and Analysis of Algorithms.
2. Lectures 4-6 of Chandan Saha's course "Topics in Complexity Theory" from Spring 2015.