EFFICIENT PARALLEL CIRCUITS AND ALGORITHMS FOR DIVISION
(U) ILLINOIS UNIV AT URBANA APPLIED COMPUTATION THEORY
GROUP N SHANKAR ET AL JUN 87 ACT-78 N8814-84-C-0149
UNCLASSIFIED 8-87
EFFICIENT PARALLEL CIRCUITS AND ALGORITHMS FOR DIVISION

Narayan Shankar
Vijaya Ramachandran
**Title:** Efficient Parallel Circuits and Algorithms for Division

**Personal Authors:** Shankar, Narayan and Ramachandran, Vijaya

**Abstract:** We improve the size bound for parallel circuits and algorithms for the division problem.
Efficient Parallel Circuits and Algorithms for Division

Narayan Shankar
Vijaya Ramachandran

Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

ABSTRACT

We improve the size bound for parallel circuits and algorithms for the division problem.

Keywords: division, boolean circuits, PRAM algorithm, efficient computation

This work was supported by the National Science Foundation under Grant ECS 8404866, the Semiconductor Research Corporation under Grant SRC 86-12-109, and the Joint Services Electronics Program under N00014-84-C-0149.
1. Introduction

In this paper we consider the problem of finding the \(n\)-bit result of dividing one \(n\)-bit number by another. We present circuits with asymptotically small size and depth for this problem and we derive from them, efficient PRAM algorithms for division in the bit model. Our primary result, which is a size-efficient implementation of the circuits in Reif [Re86], is a logspace uniform family of circuits for division of depth \(D(n)=O(\text{log} \cdot \text{loglog} n)\) and size \(S(n)=O((1/8^\delta \cdot n^{1+\delta}))\), for any \(\delta>0\). This translates into a uniform parallel algorithm on a shared memory machine (PRAM) with bit operations and exclusive memory writes with parallel time \(D(n)\) using \(O(S(n))\) processors. It also translates into a parallel algorithm for a concurrent write PRAM with parallel time \(O(D(n)/\text{loglog} n)\) using \(O(S(n))\) processors. Finally, we apply the results of Beame, Cook and Hoover [BeCoHo86] to obtain a polynomial-time uniform family of circuits for division of depth \(O((1/8^\delta \cdot \text{log} n))\) and size \(O(S(n))\).

2. Parallel Models of Computation

A (bounded fan-in) boolean circuit is an acyclic labeled digraph. Nodes are labeled as input, constant, AND, OR, NOT, or output nodes. Input and constant nodes have zero fan-in, AND and OR nodes have fan-in of 2, NOT and output nodes have fan-in of 1. Output nodes have fan-out zero.

Let \(B=(0,1)\). A boolean circuit with \(n\) input and \(m\) output nodes computes a boolean function \(f:B^n\rightarrow B^m\). The size of a boolean circuit is the number of nodes in the circuit excluding input and output nodes. The depth of the circuit is the length of the longest path among all paths from input to output nodes. Given a sequence of circuits \(C_1, C_2, \cdots\) we denote the size of the \(n\)-th circuit by \(\text{SIZE}(C_n)\) and its depth by \(\text{DEPTH}(C_n)\). If there exists a function \(S(n)\) such that \(\text{SIZE}(C_n)\leq S(n)\) for each \(n\) then we say that the size of the sequence is \(O(S(n))\). Similarly we may define the depth \(O(D(n))\) of the sequence. We say a sequence is in \(\text{SIZE-DEPTH}(S(n), D(n))\) if it is simultaneously bounded in size by \(O(S(n))\) and in depth by \(O(D(n))\). A sequence of boolean functions will be referred to as a problem, and a sequence of circuits such that the \(n\)-th circuit realizes the \(n\)-th boolean function is an algorithm to solve the problem. We will say that an algorithm gives circuits of small size for a problem if \(S(n)=O(f(8^\delta \cdot n^{1+\delta}))\), for any \(\delta>0\) and for some function \(f\). Small size circuits are desirable since they lead to low hardware costs. For parallel computation, we also want \(D(n)\) to be small, since \(D(n)\) gives the parallel computation time.
A sequence of circuits is logspace uniform for a problem if there exists a logspace-bounded turing machine that computes a suitable binary encoding of the $n$-th circuit on being input the number $n$ in unary. For theoretical reasons logspace uniformity is a desirable property in a sequence of circuits (see, e.g., [Ru81]).

A PRAM in the bit model is a parallel RAM with access to a global memory and with each processor capable of a bit operation in unit time. A CREW PRAM is a PRAM allowing concurrent reads but only exclusive writes on the global memory. A CRCW PRAM allows concurrent reads and writes. A PRAM algorithm is uniform if the algorithm is parametrized by $n$ and works in the parallel time bound for all values of $n$.

3. Previous Work on Circuits for Basic Arithmetic Operations

For the problem of adding two $n$-bit numbers Krapchenko [Kr70], and Ladner and Fischer [LaFi80] present algorithms which achieve asymptotically optimal delay of $\log n$ and a linear size bound which are the best possible in this model (see [Sa76]).

For adding $n$ $n$-bit numbers Ofman [Of62] and Wallace [Wa64] have $O(\log n)$ depth circuits with $O(n^2)$ gates, which is linear in the size of the input.

The best known circuits for the multiplication of two $n$-bit numbers are due to Schonhage and Strassen [ScSt71]. These have $O(\log n)$ depth and $O(n \log n \log \log n)$ size.

In the division problem, we need to compute the $n$-bit quotient $u/v$ where $u$ and $v$ are $n$-bit numbers. Since $u/v = u(1/v)$, and multiplication can be done efficiently by the Schonhage-Strassen algorithm, attention has been concentrated on the computation of the $n$-bit reciprocal of an $n$-bit number. The first good circuits for the reciprocal problem are due to Cook [Co66]. The method used in [Co66] to compute the reciprocal is to first normalize $v$ to a number in the interval $(1/2, 1)$, set $x = 1 - v$, and compute $1/(1-x) = 1 + x + x^2 + x^3 + \ldots$ where the first $n$ terms of the series give sufficient precision. The problem is thus reduced to that of efficiently computing $x^n$ where $x$ is an $n$-bit number. [Co66] presents a $O(\log^2 n)$ depth polynomial size family of circuits for this problem. Recent attempts to obtain better circuits for division have all concentrated on the powering problem.
A log depth polynomial size circuit for division has been obtained in [BeCoHo86], which solves the powering problem by using a combination of Chinese remaindering and taking logarithms over finite fields. The algorithm does not appear to be logspace uniform, though the circuits are polynomial time computable. Reif [Re86] has logspace uniform $O(\log n \log \log n)$ depth division circuits of polynomial size, parametrized only by $n$, so that the algorithm translates into a uniform CREW PRAM algorithm. The division circuits in both [Re86] and [BeCoHo86] are worse than quadratic in size. Thus while there are known small size circuits for addition and multiplication of $O(\log n)$ depth, this is not the case for division.

In the next section we present circuits of size $O((1/\delta^4) \cdot n^{1+\delta})$, for any $\delta > 0$, that achieve the same depths as in [Re86] and [BeCoHo86], thus obtaining small size circuits of small depth for the division problem.

We close this section with a brief discussion of the DFT, presenting the definitions and theorems that we need (for further details and proofs see [AhHoUl74])

Let $R$ be a commutative ring with identity 1. Then the set of all infinite sequences from $R$ with only finitely many non-zero terms forms a commutative ring with identity 1 under componentwise addition, and multiplication defined by convolution. This ring is called the ring of formal polynomials over $R$ and is denoted by $R[t]$, and the sequence whose $(k+1)$-th term is non-zero and whose later terms are all zero is denoted simply by $(a_0, a_1, \ldots, a_k)$ and also often written as $a_0 + a_1 t + \cdots + a_k t^k$.

Let $R$ have a primitive $n$-th root of 1 (denoted by $\omega$), where $n$ is a unit in $R$, i.e. $n$ has an inverse in $R$. Let the $n \times n$ matrix $M = (m_{ij})$ be defined by $m_{ij} = \omega^{ji}$, $i=0,1,\ldots,n-1$. The matrix $M$ is invertible. If $A$ is an $n$-vector, define $DFT_n(A) = MA$ and $DFT_n^{-1}(A) = M^{-1} A$. Note that $DFT_n^{-1}(DFT_n(A)) = A$. [CoTu65] introduced an algorithm which translates into a size $O(n^2 \log n)$, depth $O(\log n)$ circuit for computing the DFT of $n$ $n$-bit numbers. If we further assume that there exists $\psi$ in $R$ such that $\psi^2 = -1$ and $\psi^3 = \omega$, then with any $(n-1)$-degree polynomial $A(t) = a_0 + a_1 t + a_2 t^2 + \cdots + a_{n-1} t^{n-1}$ we can associate $A^* = (a_0, a_1, \psi a_2, \ldots, a_{n-1} \psi^{n-1})$. We now state

The negatively wrapped convolution theorem: Let $A_i(t)$, $i=1,2,\ldots,r$ be $(n-1)$-degree polynomials, and $B(t)$

$= \prod_{i=1}^r A_i(t)$, this being the ring product in $R[t]$. Let $D(t) = B(t) \mod t^n + 1$. Then $D^* = DFT_n^{-1}(\prod_{i=1}^r DFT_n(A_i^*))$
where the multiplication in the transformed domain is componentwise.

4. The Reciprocal Problem

Let $x$ be an $n$-bit number in $[1/2, 1)$. We wish to compute $v=1/x$ correct to $n$ places to the right of the point. Let $u=1-x$, where $0<u<1/2$ and $u$ has $n$ bits. Then $1/x=1/(1-u)=1+u+u^2+u^3+\ldots$. We would obtain $1/x$ to sufficient precision if we compute $1,u,u^2,u^3,\ldots,u^{n-1}$ exactly (each of these numbers can be represented exactly using at most $n^2$ bits since $u$ has only $n$ bits), and add them up and truncate the sum to $n$ bits to the right of the point. Since in this method the powers are computed to $n^2$ bits of precision, the resulting circuit has size $\Omega(n^2)$. However, the computation of the powers to $n^2$ bits is unnecessary, as is shown in [MePr86]. Consider the factorization:

$$1+u+u^2+\ldots+u^{n-1}=(1+u+u^2+\ldots+u^{s-1})(1+u^2+u^4+\ldots+u^{(s-1)/2})\ldots(1+u^{s-1}+\ldots+u^{(s-1)s^{n-1}})$$

where $s=n^{1/m}$, $m$ being a fixed integer. Denote the $i$-th factor $(1+u^{i-1}+u^{2i-1}+\ldots+u^{(s-1)i^{i-1}})$, by $\phi_i^{\prime}$, $i=0,1,\ldots,(m-1)$. We can compute each factor $\phi_i^{\prime}$, and then multiply the $m$ factors and truncate the result to $n+2$ bits to get $1/x$. Note that $\phi_i^{\prime}$ is actually the sum of the powers $u^j$ for $j=0,1,2,\ldots,(s-1)$. It is proved in [MePr86] that if we use an $n+\log(12m)$ bit approximation to $u^j$ (which we denote by $\theta_j$), compute $\phi_j$, $j=0,1,\ldots,s-1$ exactly, i.e. to $ns$ bits, and add these $s$ numbers to get an $ns$-bit approximation $\phi_i$ to $\phi_i^{\prime}$, then the product of the $m \phi_i$ truncated to $n$ bits gives an $n$-bit approximation to $1/x$. The $\phi_i$ are obtained as follows: $\theta_0$ is initialized to $u$, and $\theta_i$ is obtained by computing $\theta^{\prime}_{i-1}$ and truncating to $n+\log(12m)$ bits. Note that no computation involves more than $ns$ bits. Though the above factorization is not valid if $n^{1/m}=s$ is not an integer, this algorithm for computing the reciprocal is still good with $s=\lceil n^{1/m} \rceil$, since this only means that we compute even more than $n$ terms in the series.

Let $S_1(s,n)$ and $T_1(s,n)$ denote the size and time of computing each $\phi_i$. If $S_0(n)$ is the size of reciprocation, then $S_0(n) = O(msS_1(s,n)+mM(ns))$, where the first term on the RHS is the cost of computing $m \phi_i$, and the second term is the cost of multiplying them together. If we denote the depth for computing reciprocal by $T_0(n)$ then $T_0(n) = O(mT_1(s,n)+\log m \log n)$, where the first term is the depth for computing $m \phi_i$ and the second term the depth for multiplying them together. Since $\phi_i$ is the sum of $\theta_j^{\prime}$, $j=0,1,\ldots,s-1$, with $\theta_j$ being of $O(n)$ bits, if we denote the size of computing the $s$-th power of an $n$-bit number by $\log s$, then the size of computing the reciprocal is $O(ns^2+\log^2 ns)$. The time for computing each $\phi_i$ is $O(ns+\log^2 ns)$, and the time for multiplying the $m$ factors is $O(n^2+\log^2 nm)$. The total time is $O(n^2+\log^2 ns)$.
number by $S_2(s,n)$, we find that $S_1(s,n)=sS_2(s,n)+ns^2$, where the first term is the cost of computing the $s$ powers of $\theta_1$ and the second term is the cost of adding them up. Similarly $T_1(s,n)=T_2(s,n)+\log ns$, where the first term is the depth of computing the $s$ powers of $\theta_1$ in parallel, and the second term the depth of adding them up. From the above it follows that

$$S_0(n)=sS_2(s,n)+nms^2+mM(ns)$$

and

$$T_0(n)=mT_2(s,n)+m\log ns$$

We now develop an efficient algorithm for computing the $s$-th power of an $n$-bit number and determine its size $S_2(s,n)$ and time $T_2(s,n)$. We actually describe an algorithm that computes the $s$-th power of an $r$-bit number modulo $2^r+1$. This will be used to compute the $s$-th power of an $n$-bit number exactly, by treating the input as an $n$-bit number and computing its $s$-th power mod $2^m+1$. Our algorithm is based on the modular product algorithm in [Re86] and uses the DFT. We first introduce some notation. The ring $\mathbb{Z}_{2^k+1}$ has $k$ as a unit, $\omega=4$ as a primitive $k$-th root of unity and $\psi=2$ satisfies $\psi^k=-1$ and $\psi^2=\omega$. Hence DFT and inverse DFT of $k$-sequences can be defined in this ring and by the notation $DFT_k(x_0,x_1,\ldots,x_{k-1})$ mod $2^k+1$ we mean the DFT (as previously defined) in the ring $\mathbb{Z}_{2^k+1}$ with $\omega=4$. Denote by $(x_0,\ldots,x_{k-1})$ mod $2^k+1$ the vector $(x_0,\ldots,x_{k-1})$ with each of its components reduced modulo $2^k+1$. We now state the algorithm and follow it up with a discussion.

The modular power algorithm

Input $x=x_{r-1}x_{r-2}\cdots x_0$ is of $r$ bits, the least significant bit being $x_0$ and the most significant bit being $x_{r-1}$. $r$ is a power of 2. Output is $x^s$ mod $2^r+1$, where $s=r^s$, $0<s<1/2$.

Function Modpower $(x,r,s)$

begin
  if $r\leq 4$
    Modpower$\leftarrow x^s$ mod $2^r+1$; (* compute directly by constant depth, constant size circuit since $r\leq 4$ and $s\leq 2$; this is the base step of recursion *)
  else
    begin
      case $r\geq 2s^2$ do
        begin
          $l\leftarrow (1/2)(\sqrt{r/s})$;
        end
      end
    end
end
divide $x$ into $k$ equal blocks of $l$ bits each and form the vector $(x_0, x_1, \ldots, x_{k-1})$ where $x_i = \{i-d-1 \cdots i-(d+1)\}$ (* this is the vector $g(i)$ in the discussion below *)

$$(x_0, x_1, x_2, \ldots, x_{k-1}) \leftarrow (x_0, x_1, 2x_2 2^2, \ldots, x_{k-1} 2^{k-1}) \mod 2^k + 1;$$

(* this is $g^*$ *)

$$(x_0, x_1, x_2, \ldots, x_{k-1}) \leftarrow DFT_k(x_0, x_1, \ldots, x_{k-1}) \mod Z_{2^k+1};$$

(* this is $DFT_k(g^*)$ *)

para do $x_i \leftarrow Modpower(x_i, k, s)$; (* this is the componentwise powering of $k$ smaller numbers in the transformed domain, done recursively in parallel *)

$$(x_0, x_1, x_2, \ldots, x_{k-1}) \leftarrow (x_0, x_1, 2x_2 2^2, \ldots, x_{k-1} 2^{k-1}) \mod 2^k + 1;$$

(* this is $d(i)$ *)

$$(x_0, x_1, x_2, \ldots, x_{k-1}) \leftarrow (x_0 + x_1 2^1 + x_2 2^2 + \ldots + x_{k-1} 2^{k-1}) \mod 2^k + 1;$$

(* this is $d(2')$, which is what we want *)

end

case $r < s^2$ do

begin

$x \leftarrow x^r \mod 2^r + 1$ (* compute by the modular product algorithm in [Re86] *)

end

end.

Remarks on the algorithm

The main idea in the algorithm is to split $x$ into $k$ blocks, $k=2^{\sqrt{s}}$, and construct the vector $g(i) = a_0 + a_1 i + a_2 i^2 + \ldots + a_{k-1} i^{k-1}$. If $l=r/k$, then $g(2') = x$. If we let $d(i) = (g(i))^r \mod i^k + 1$, then $d(2') = (g(2'))^r \mod (2^k + 1) = x^r \mod 2^r + 1$. Finding $d(i)$ would solve the problem, since its value at $2'$ is the desired $x^r \mod 2^r + 1$. The polynomials above are over the ring of integers $\mathbb{Z}$ but we do calculations over the finite ring $Z_{2^k+1}$. Apply the convolution theorem to get $d^* = DFT_k^{-1}(DFT_k(g^*)) \mod 2^k + 1$, where the powering is componentwise in the transformed domain. Notice that there are $k$ powerings in the transformed domain, each of $k$-bit numbers mod $2^k + 1$, where $k$ is smaller than $r$, and we do these powerings recursively. We need to be sure that the $d$ computed as above (in $\mathbb{Z}$) gives the correct $d$ (in $\mathbb{Z}$). This will be so if the coefficients of $d$ are small enough i.e., $\leq 2^{s-1}$. This can be arranged by requiring $s = r^e$, $0 < e \leq 1/2$, and by choosing $k = 2^\sqrt{s}$ as we have done. (It is shown in [Re86] that it is sufficient to choose $s$ and $k$ so that $2s((r/k+1+\log k)) \leq k - 1$ is satisfied. The above choices of $s$ and $k$ satisfy this inequality for sufficiently large $r$.) Essentially what we are doing is making each coefficient of $g$ small enough by splitting $x$ into sufficiently many pieces. Since the ring $Z_{2^k+1}$ grows with $k$ (the number of coefficients in $g$), if we have a sufficiently large number of coefficients and we make the power $s$ small enough, we can expect $g^*$ to have small enough coefficients, so that it can be represented without error in $Z_{2^k+1}$. 
Choice (1) of the case statement of the algorithm is entered recursively \( \log(1/e) \) times on first calling the program. We start with \( k=r \) and in each subsequent application \( k=2\sqrt{k-1} \), and we keep going until \( k<s^2 \). We set up the recurrence \( k_0=r, k_2=2\sqrt{k-1} \), and solve to get \( k=(4s)^{1-\frac{1}{2\sqrt{r}}} \). In about \( \log(1/e) \) steps \( k_1<s^2 \).

At this point we need the \( s \)-th power of an \( s^2 \)-bit number. Now choice (2) of the case statement is executed. Since \( s=r^e \) is small compared with \( r \), we do not attempt to be efficient with this residual computation, but simply apply the algorithm in [Re86]. The size complexity of the algorithm is dominated by choice (1) of case, as our analysis will show.

5. Gate Count

As already mentioned the exact value of \( x^e \) may be found by treating \( x \) as an \( ns \)-bit number and computing \( x^e \) mod \( 2^{2n}+1 \). This is accomplished by calling \textit{Modpower}(x,ns,s). We now compute the size and depth for this problem. We solved a recurrence for \( k_i \) above with initial condition \( k_0=r \). With initial condition \( k_0=ns \) (which is the case in the exact powering problem), \( k_i=4^{1-\frac{1}{2\sqrt{r}}} n^{\frac{1}{2\sqrt{r}}} \). Recall that we denoted by \( S_2(s,n) \) the size needed for computing the \( s \)-th power of an \( n \)-bit number. If \( S_s(n) \) denotes the size of computing the \( s \)-th power of an \( n \)-bit number mod \( 2^{2n}+1 \) then \( S_2(s,n)=S(s,ns) \) as seen above.

From choice (1) of case in the algorithm we obtain

\[
S(s,ns)=S(s,k_0)=ck_1^2 \log k_1 + k_1 S_s(s,k_1)
\]

The first term on RHS is the cost of taking DFT of a \( k \)-vector in \( Z_{2^{2n}+1} \) by the Cooley-Tukey algorithm [CoTu65]. The second term is for the recursive computation of \( k_1 \) smaller powerings. (Computing \( g^* \) from \( g \) and \( d \) from \( d^* \) in the algorithm needs \( O(k_1) \) per entry and hence \( O(k_1^2) \) overall, which follows from lemma 7.6 pp 266 [AhHoUl74]. Computing \( g(2^i) \) is like adding two \( n \)-bit numbers, and needs only size \( O(k_1^2) \). These steps in the algorithm are all dominated by the cost of computing the DFT.) Now replace \( S(s,k_1) \) by an expression in terms of \( k_2 \) and keep doing this for \( l \) steps to get

\[
S(s,k_0)=c \sum_{i=1}^{l} \left( \prod_{i=1}^{i} k_i \right) k_i \log k_i + (\prod_{i=1}^{l} k_i) S(s,k_i)
\]

Using the formula for \( k_i \) we get

\[
\prod_{i=1}^{l} k_i = 4^{1-\frac{1}{2\sqrt{r}}} \cdot n^{\frac{1}{2\sqrt{r}}}
\]
Also we have

\[
\left(\prod_{i=1}^{\infty} k_i\right)k_r=4^r n r^{r+1}
\]

and \(\log k_i \leq c \log n\) since \(s=n^\varepsilon\) with \(\varepsilon \leq 1/2\). This gives

\[
S(s,k_0)=c^\varepsilon \log n \left(\sum_{i=1}^{\infty} \prod_{j=1}^{i} k_j\right) + \left(\prod_{i=1}^{\infty} k_i\right) S(s,k_i)
\]

Using our previous formula for \(\prod_{j=1}^{i} k_j\) we get

\[
S(s,k_0)=c^\varepsilon n^4 l s^{1+1} \log n + 4^{1+1+1/2} n^{-1/2} l s^1 S(s,k_i)
\]

With \(l=\log(1/e)\) and \(s=n^\varepsilon\) this becomes

\[
S(n^\varepsilon,n^{1+\varepsilon})=c^\varepsilon (l/e^2) n^{1+\varepsilon+\log(1/e)} \log n + (l/e^2) n^{-1+\varepsilon+\log(1/e)} S(n^\varepsilon,n^{2\varepsilon})
\]

\(S(n^\varepsilon,n^{2\varepsilon})\) is the size complexity of choice (2) of case, which is \(O(n^{4\varepsilon})\). This gives us

\[
S_2(n^\varepsilon,n)=S(n^\varepsilon,n^{1+\varepsilon})=c^\varepsilon (l/e^2) n^{1+3\varepsilon+\log(1/e)}
\]

Using this in equation (1) we find that the size for computing reciprocal is

\[
S_0(n^\varepsilon,n)=c^\varepsilon (l/e^2) n^{1+4\varepsilon+\log(1/e)}
\]

(We have omitted the second term on the RHS of eqn (1) since it is dominated by the first.) From this it follows that for any \(\delta>0\), there are circuits for computing the reciprocal that have size \(O((1/\delta^4)n^{1+\delta})\). Suppose \(\delta>0\) is given. We solve for \(\varepsilon\) using the equation \(\delta=4\varepsilon+\varepsilon \log(1/\varepsilon)\), and construct circuits as above with this \(\varepsilon\). Clearly \(\varepsilon<\delta\) and since for small \(\varepsilon, \delta<e^{3\delta}\), the above result is true.

Recall that we denoted by \(T_2(n^\varepsilon,n)\) the depth for computing the \(n^\varepsilon\)-th power of an \(n\)-bit number. Note that choice (1) of case is entered \(\log(1/e)\) times, and each application is dominated in depth by the DFT computation. The total contribution from all this to the depth is \(O(\log(1/e)\log n)\). As for choice (2), the \([Re86]\) algorithm needs \(O(\log n \log \log n)\) depth to compute the \(r\)-th power of an \(r\) bit number mod \(2^r+1\). Using this with \(r=n^\varepsilon\) we see that choice (2) of case contributes \(O(\varepsilon \log n \log \log n)\) to the depth. Hence we have

\[
T_2(n^\varepsilon,n)=\log(1/e)\log n + e \log n \log \log n
\]

Using this in equation (2) we get the depth complexity of division

\[
T_0(n)=(1/e)(\log(1/e))\log n + \log n \log \log n = O(\log n \log \log n)
\]
We observe that the additional factor loglogn for the depth in the above algorithm arises from the final application of the [Re86] algorithm directly in choice (2) of case. We can avoid this at the cost of losing logspace uniformity by using one application of the [BeCoHo86] algorithm to complete the computation once we get to the point where we need to compute the n^ε-th power of an n^2ε-bit number. Until this point we have used depth of log(1/ε)logn and with the additional depth of elogn of the [BeCoHo86] algorithm, the n^ε-th power of an n-bit number can be computed in depth O(log(1/ε)logn). Substituting this in place of T2(n^ε, n) in equation (2) we find that the depth of the division algorithm incorporating the [BeCoHo86] circuit is O((1/ε)log(1/ε)logn) = O((1/δ^2)logn). The size of this application of the [BeCoHo86] circuit is O(n^4ε), so the size result previously obtained for the algorithm based purely on [Re86] is not affected.

For PRAM implementation, we note that Modpower is parametrized only by r and is logspace uniform, so our division circuit translates to a O(lognloglogn) time algorithm on the CREW PRAM (with bit operations) using O((1/8^4)n^1+δ) processors, for any δ>0.

Using standard techniques, (see [ChStVi84]), our logspace uniform O(lognloglogn) depth division circuit can be compressed in depth to O((1/k)logn) for any k>0, with an increase in size of a factor of 2^{log^4}. We let k=1/2 and translate the resulting unbounded fan-in circuits to a CRCW PRAM algorithm in the bit model running in time O(logn) and using O((1/δ^4)n^{1+δ}) processors, for any δ>0.

REFERENCES


END
0-87
DTIC