Using Smoothness to Achieve Parallelism (abstract)

Leonard M. Adleman * Kireeti Kompella *

Department of Computer Science
University of Southern California

1 Introduction

Currently, there is a great deal of excitement about research in the area of parallel algorithms. Considerable progress has been made in many areas: graph theory, combinatorics, matrix theory, numerical analysis, etc. For many problems NC algorithms have been found (e.g. maximal independent set [KW], [GS], sorting [AKS], permutation group membership [Lu], CFL recognition [Ru], matrix arithmetic [Cs], [PR1]). For many other problems proofs of P-completeness have been obtained (e.g., the circuit value problem [La], unification [DKM], linear programming [DLR], maximum network flow [GSS], lexicographically first maximal clique [Co]). Regrettably, the situation in computational number theory is less satisfactory. While NC algorithms have been discovered for the basic arithmetic operations [Sa], [Re], [BCH], and progress has been made on problems involving polynomials [Eb], [PR2], [BGH], the parallel complexity of such fundamental problems as integer gcds and modular exponentiation has remained open since first being raised in a paper by Cook [Co]. Indeed, it seems possible that the prospects for parallelism in computational number theory rest on these problems: if integer gcd and modular exponentiation were shown to be in NC, the prospects for finding efficient parallel algorithms for other number theoretic problems would be substantial; if, on the other hand, integer gcd and modular exponentiation were shown to be P-complete, the prospects would be considerably diminished.

In attacking these problems, two approaches seem natural: the more practical approach is to insist on a polynomial bound on the number of processors, and then try to obtain the best time; perhaps the more theoretical approach is to insist on a polylog bound on the time, and then try to obtain the best processor count. The former approach has been taken by Brent and Kung [BK], Kannan, Miller and Rudolph [KMR], and Chor and Goldreich [CG] who achieve running times of n, $n \log\log n/\log n$, $n/\log n$ respectively, while preserving a polynomial number of processors. In this paper the latter approach is taken.

The main idea in attacking the modular exponentiation problem is to extend the well known sequential method of "doubling up" when raising a number to a high power. Doubling up allows one to double the current exponent at each iteration, but is too slow to yield a polylog algorithm for modular exponentiation. Instead, the algorithm presented here uses "squaring up". That is, on each iteration the current exponent is squared. Similarly, the Euclidean Algorithm is too slow to yield a polylog algorithm for the gcd of two numbers. The algorithm presented here uses a strategy which produces the gcd of two numbers in essentially one parallel step. These ideas lead to polylog depth, subexponential size circuits for integer gcd and modular exponentiation. More formally:

Theorem 1 *There exists a logspace uniform family*

*Research supported under NSF Grant # CCR 8519296

of probabilistic circuits $< \alpha_n >$ and a $c \in \mathcal{Z}_{>0}$ such that:

1. *For all $n, a, b, m \in \mathcal{Z}_{>0}$ with m a prime, and $|a|^1 + |b| + |m| = n$, α_n on input $< a, b, m >$ produces an output with probability $> 1/2$.*
2. *For all $n, a, b, m \in \mathcal{Z}_{>0}$ with m a prime, and $|a| + |b| + |m| = n$, if α_n on input $< a, b, m >$ produces an output, then the output is $a^b \bmod m$.*
3. *$D_{mexp} = O(\log^3 n)$ where for all $n \in \mathcal{Z}_{>0}$, $D_{mexp}(n)$ =depth of α_n.*
4. *$S_{mexp} = O(e^{c\sqrt{n \log n}})$ where for all $n \in \mathcal{Z}_{>0}$, $S_{mexp}(n)$ =size of α_n.*

The general case (i.e. when the modulus m is arbitrary) will be established in a subsequent paper.

Theorem 2 *There exists a logspace uniform family of probabilistic circuits $< \alpha_n >$ and a $c \in \mathcal{Z}_{>0}$ such that:*

1. *For all $n, a, b \in \mathcal{Z}_{>0}$ with $|a| + |b| = n$, α_n on input $< a, b >$ produces an output with probability $> 1/2$.*
2. *For all $n, a, b \in \mathcal{Z}_{>0}$ with $|a| + |b| = n$, if α_n on input $< a, b >$ produces an output, then the output is gcd$(a, b), x, y$ such that $ax - by = \gcd(a, b)$.*
3. *$D_{gcd} = O(\log^2 n)$ where for all $n \in \mathcal{Z}_{>0}$, $D_{gcd}(n)$ =depth of α_n.*
4. *$S_{gcd} = O(e^{c\sqrt{n \log n}})$ where for all $n \in \mathcal{Z}_{>0}$, $S_{gcd}(n)$ =size of α_n.*

The function $e^{\sqrt{n \log n}}$ arises as a result of the use of "smooth" numbers, which will be defined below. Smooth numbers have been used to reduce the sequential time complexity of certain important problems (e.g. integer factoring and discrete logarithms) from naive exponential bounds to strictly subexponential bounds. It appears that for many problems, smooth numbers can also be used to reduce the circuit size complexity from naive exponential bounds to strictly subexponential bounds. Proofs for the following will be outlined:

Theorem 3 *There exist a logspace uniform family of probabilistic circuits $< \alpha_n >$ and a $c, d \in \mathcal{R}_{>0}$ such that:*

1. *For all $n, m \in \mathcal{Z}_{>0}$ with m composite and $|m| = n$, α_n on input m produces an output with probability $> 1/2$.*
2. *For all $n, m \in \mathcal{Z}_{>0}$ with m prime and $|m| = n$, α_n on input m does not produce an output.*
3. *For all $n, m \in \mathcal{Z}_{>0}$ with $|m| = n$, if α_n on input m produces an output, then the output is 1.*
4. *$D_{composite} = O(\log^d n)$ where for all $n \in \mathcal{Z}_{>0}$, $D_{composite}(n)$ =depth of α_n.*
5. *$S_{composite} = O(e^{c\sqrt{n \log n}})$ where for all $n \in \mathcal{Z}_{>0}$, $S_{composite}(n)$ =size of α_n.*

Theorem 4 *There exist a logspace uniform family of probabilistic circuits $< \alpha_n >$ and a $c, d \in \mathcal{R}_{>0}$ such that:*

1. *For all $n, m \in \mathcal{Z}_{>0}$ with $|m| = n$ α_n on input m with m composite produces an output with probability $> 1/2$.*
2. *For all $n, m \in \mathcal{Z}_{>0}$ with $|m| = n$, if α_n on input m with m composite produces an output f, then $1 < f < m$ and $f | m$.*
3. *$D_{fact} = O(n^d)$ where for all $n \in \mathcal{Z}_{>0}$, $D_{fact}(n)$ =depth of α_n.*
4. *$S_{fact} = O(e^{c\sqrt{n \log n}})$ where for all $n \in \mathcal{Z}_{>0}$, $S_{fact}(n)$ =size of α_n.*

Theorem 5 *There exist a logspace uniform family of probabilistic circuits $< \alpha_n >$ and a $c, d \in \mathcal{R}_{>0}$ such that:*

1. *For all $n, a, b, m \in \mathcal{Z}_{>0}$ with $|a| + |b| + |m| = n$, α_n on input a, b, m such that there exists an $x \in \mathcal{Z}_{>0}$ with $a^x \equiv b \bmod m$ produces an output with probability $> 1/2$.*
2. *For all $n, a, b, m \in \mathcal{Z}_{>0}$ with $|a| + |b| + |m| = n$, if α_n on input a, b, m such that there exists an $x \in \mathcal{Z}_{>0}$ with $a^x \equiv b \bmod m$ produces an output f then $a^f \equiv b \bmod m$*
3. *$D_{dlog} = O(n^d)$ where for all $n \in \mathcal{Z}_{>0}$, $D_{dlog}(n)$ =depth of α_n.*
4. *$S_{dlog} = O(e^{c\sqrt{n \log n}})$ where for all $n \in \mathcal{Z}_{>0}$, $S_{dlog}(n)$ =size of α_n.*

[1] Throughout, $|x|$ denotes the length of x written in binary

For each problem the depth of the corresponding circuit is a polynomial in the log of the running time of the best known sequential solution. Although smoothness is used in achieving these results, its use varies greatly from problem to problem.

Finally, some relations between problems will be established. Let

- Primality Testing be the problem of computing the function f such that for all $x \in \mathcal{Z}_{>0}$ if x is prime f(x)=1, if x is composite f(x)=0.
- Compositeness Testing be the problem of computing the function f such that for all $x \in \mathcal{Z}_{>0}$ if x is composite f(x)=1, if x is prime f(x) is undefined.
- Relative Primality be the problem of computing the function f such that for all $a, b \in \mathcal{Z}_{>0}$ if (a,b)=1 then f(a,b)=1, if (a,b) /=1 then f(a,b)=0.

Theorem 6

1. *Relative Primality $\leq_{RNC}$ Primality Testing.*
2. *Compositeness Testing $\leq_{RNC}$ Modular Exponentiation.*

(Part 1 of this theorem has been independently produced by Karloff [Bor]; and Part 2 by Fich and Tompa [FT]).

2 Modular Exponentiation

Consider the problem MODEXP(a, b, m) of computing $a^b \bmod m$ for integers a, b and m. In this paper, only the case when m is a prime will be considered. Previous work has shown that: if the exponent b is bounded by $O(\log m)$ then this problem is in NC^1(P-uniform) [BCH]; if the modulus m is a product of primes $\leq \log m$, then the problem is in NC^2(P-uniform) [vzG]. In the general case, however, there is no known NC algorithm for modular exponentiation; nor is there a proof of the problem being P-complete.

To begin, consider the problem 2^{2^f}POWERS(a,m): when m is prime compute $a^{2^{2^f}} \bmod m$ for $0 \leq f \leq \lceil \log\log m \rceil$. A parallel algorithm for 2^{2^f}POWERS will be presented. The main idea used in this algorithm, "squaring up", can be illustrated as follows: Assume that $a^{2^{2^f}} \bmod m$ is known and $a^{2^{2^{f+1}}}$ nod m is to be computed in the next iteration. For the purpose of exposition, assume that $a^{2^{2^f}} \bmod m$ is the square free product of small primes, i.e. $a^{2^{2^f}} \bmod m = \prod p_l$ with the p_l small. Assume further that $p_l^{2^{2^f}} \bmod m$ is known for each l. Then the following holds:

$$a^{2^{2^{f+1}}} = (a^{2^{2^f}})^{2^{2^f}} \equiv (\prod p_l)^{2^{2^f}} \equiv \prod p_l^{2^{2^f}} \bmod m,$$

and so $a^{2^{2^{f+1}}} \bmod m$ can be obtained by just a few parallel multiplications. This idea is the basis for parallelizing 2^{2^f}POWERS.

Of course, $a^{2^{2^f}} \bmod m$ will not in general have only small prime factors. Before dealing with this case, consider the following:

Definition 1 *Let $B \in \mathcal{R}_{>0}$ and $x \in \mathcal{Z}_{>1}$. Then x is B-smooth iff for all $p \in \mathcal{Z}_{>1}$ with p prime and $p|x$, $p \leq B$.*

Let $L : \mathcal{R}_{>0} \rightarrow \mathcal{R}_{>0}$ be defined by $L(x) = e^{\sqrt{\log x \log\log x}}$. For all $m \in \mathcal{Z}_{>1}$ let $\Psi'(m, L(m)) = \#\{z \mid z \leq m, z \text{ squarefree and } z \text{ is } L(m)\text{-smooth}\}$.

The following elementary theorem, whose proof is omitted, will be used:

Theorem 7 *There exists a $c_s \in \mathcal{Z}$ such that for all $m \in \mathcal{Z}_{>1}$*

$$\Psi'(m, L(m)) \geq m/L(m)^{c_s}$$

To apply the method of "squaring up" in the general case, assume that a large number of "helper" numbers r_k are available, whose inverses mod m, denoted s_k, have been computed. Assume also that $p_l^{2^{2^f}} \bmod m$ are known for $p_l \leq B$ (the smoothness bound), and $s_k^{2^{2^f}} \bmod m$ for all k. The modified idea then goes as follows: For each r_k, compute $t = a^{2^{2^f}} r_k \bmod m$. If t is B-smooth, say $\prod q_l$, q_l prime, then

$$a^{2^{2^{f+1}}} = (a^{2^{2^f}})^{2^{2^f}} \equiv (a^{2^{2^f}} r_k s_k)^{2^{2^f}} \equiv (t)^{2^{2^f}} s_k^{2^{2^f}}$$

$$\equiv (\prod q_l)^{2^{2^f}} s_k^{2^{2^f}} \equiv (\prod q_l^{2^{2^f}}) s_k^{2^{2^f}} \pmod m$$

and so, once again, $a^{2^{2^f}} \bmod m$ can be computed with just a few parallel multiplications.

In the same manner, $p^{2^{2^{j+1}}} \bmod m$ for all primes $p \leq B$ can be computed. Finally, the "helpers" are used to "help" each other in a similar manner.

2.1 Algorithm 2^{2^f}POWERS

An algorithm to compute 2^{2^f}POWERS(a, m) is described below. The algorithm runs on a CRCW PRAM with arbitrary write-conflict resolution [see GR]: when several processors attempt to write into the same location, one of them, arbitrarily chosen, will succeed. Let $B = \lceil L(m) \rceil$, and $N = \lceil L(m)^{c_s} 3 \log m \rceil$ where c_s is as in thereom 7. The following processors are used:

- $PR_{i,j}$ $1 \leq i, j \leq B$.
- $H_{j,h}$ $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$.
- $P_{i,j,h,k}$, $1 \leq i \leq B$, $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$, $1 \leq k \leq B$.
- $PA_{i,j,h}$, $1 \leq i \leq B$, $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$.
- $TP_{i,j,h}$, $1 \leq i \leq B$, $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$.
- $MPI_{i,j,h,l}$ $1 \leq i \leq B$, $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$, $1 \leq l \leq \lceil \log m \rceil^2$.
- $S_{i,g,j,h,k}$, $1 \leq i, j \leq N$, $1 \leq g, h \leq \lceil \log m \rceil$, $1 \leq k \leq B$.
- $SA_{i,g,j,h}$, $1 \leq i, j \leq N$, $1 \leq g, h \leq \lceil \log m \rceil$.
- $TS_{i,g,j,h}$, $1 \leq i, j \leq N$, $1 \leq g, h \leq \lceil \log m \rceil$.
- $MSI_{i,g,j,h,l}$ $1 \leq i, j \leq N$, $1 \leq g, h \leq \lceil \log m \rceil$, $1 \leq l \leq \lceil \log m \rceil^2$.
- $A_{j,h,k}$, $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$, $1 \leq k \leq B$.
- $AA_{j,h}$, $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$.
- $TA_{j,h}$, $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$.
- $MAI_{j,h,l}$ $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$, $1 \leq l \leq \lceil \log m \rceil^2$.

The following variables are used:

- a, b, m — input, with m a prime;
- prime[i], $2 \leq i \leq B$ — boolean
  Initialized to *true*.
- prime[1], — boolean
  Initialized to *false*.
- prpow[i,j], $1 \leq i \leq B$, $1 \leq j \leq \lceil \log m \rceil$
  Intended to hold i^j.
- r[j,h], $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$
  Initialized to random number in range $1, 2, ..., m$.
  Intended to hold $r_{j,h}$ relatively prime to m.
- s[j,h], $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$
  Intended to hold $r_{j,h}^{-1}$.
- pza[i,j,h] $1 \leq i \leq B$,$1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$.
- pzb[i,j,h] $1 \leq i \leq B$,$1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$.
- ppower[i,f], $1 \leq i \leq B$ $0 \leq f \leq \lceil \log \log m \rceil$
  Intended to hold $i^{2^{2^f}}$ mod m.
- mp[i,j,h] $1 \leq i \leq B$,$1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$.
- mpa[i,j,h,l] $1 \leq i \leq B$,$1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$, $1 \leq l \leq \lceil \log m \rceil^2$.
- mpb[i,j,h,l] $1 \leq i \leq B$,$1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$, $1 \leq l \leq \lceil \log m \rceil^2$.
- sza[i,g,j,h] $1 \leq i, j \leq N$, $1 \leq h, g \leq \lceil \log m \rceil$.
- szb[i,g,j,h] $1 \leq i, j \leq N$, $1 \leq h, g \leq \lceil \log m \rceil$.
- spower[i,h,f], $1 \leq i \leq N$, $0 \leq f \leq \lceil \log \log m \rceil$, $1 \leq h \leq \lceil \log m \rceil$.
  Intended to hold $s_{i,h}^{2^{2^f}}$ mod m.
- ms[i,g,j,h] $1 \leq i, j \leq N$, $1 \leq h, g \leq \lceil \log m \rceil$.
- msa[i,g,j,h,l] $1 \leq i, j \leq N$, $1 \leq h, g \leq \lceil \log m \rceil$, $1 \leq l \leq \lceil \log m \rceil^2$.
- msb[i,g,j,h,l] $1 \leq i, j \leq N$, $1 \leq h, g \leq \lceil \log m \rceil$, $1 \leq l \leq \lceil \log m \rceil^2$.
- aza[j,h] $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$.
- azb[j,h] $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$.
- apower[f], $0 \leq f \leq \lceil \log \log m \rceil$
  Intended to hold the output $a^{2^{2^f}}$ mod m.
- ma[j,h] $1 \leq j \leq N$, $1 \leq h \leq \lceil \log m \rceil$.

- maa[j,h,l] $1 \le j \le N$, $1 \le h \le \lceil \log m \rceil$, $1 \le l \le \lceil \log m \rceil^2$.
- mab[j,h,l] $1 \le j \le N$, $1 \le h \le \lceil \log m \rceil$, $1 \le l \le \lceil \log m \rceil^2$.

```
1. (* Compute $a^2$ *)
   a[0] := $a^2$ mod $m$

2. (* Compute inverses *)
   all processors $H_{j,h}$ $1 \le j \le N$, $1 \le h \le \lceil \log m \rceil$.
      (* Algorithm GCD is required for this step *)
      if (r[j,h],m) = 1 then s[j,h] := r[j,h]$^{-1}$ mod m
      else r[j,h] := 1 and s[j,h] := 1
      spower[j,h,0] := s[j,h]$^2$ mod m

3. (* Identify primes and compute squares *)
   all processors $PR_{i,j}$ $1 \le i, j \le B$.
      if j < i and j | i then prime[i] := *false*
      ppower[i,0]=i$^2$ mod m.

4. (* Find small prime powers *)
   all processors $PR_{i,j}$ $1 \le i \le B$, $1 \le j \le \lceil \log m \rceil$
      compute prpow[i,j] := i$^j$.

5. (* Compute the $2^{2^f}$-th powers $1 \le f \le \lceil \log\log m \rceil$ *)
   for f := 0 to $\lceil \lceil \log\log m \rceil \rceil - 1$ do
   5.a all processors $MPI_{i,j,h,l}$, $1 \le i \le B$, $1 \le j \le N$,
          $1 \le h \le \lceil \log m \rceil$, $1 \le l \le \lceil \log m \rceil^2$.
      mpa[i,j,h,l] := 1 and mpb[i,j,h,l] := 1
   5.b all processors $TP_{i,j,h}$, $1 \le i \le B$, $1 \le j \le N$.
          $1 \le h \le \lceil \log m \rceil$
      mp[i,j,h] := ppower[i,f]*r[j,h] mod m.
   5.c all processors $P_{i,j,h,k}$, $1 \le i \le B$, $1 \le j \le N$,
          $1 \le h \le \lceil \log m \rceil$, $1 \le k \le B$.
      if prime[k] and $k|mp[i,j,h]$ then
          calculate largest e such that
              prpow[k,e]|mp[i,j,h].
          choose a random number v
              in the range $1, 2, ..., \lceil \log m \rceil^2$.
          mpa[i,j,h,v] :=k and mpb[i,j,h,v] :=e
   5.d all processors $PA_{i,j,h}$, $1 \le i \le B$, $1 \le j \le N$,
          $1 \le h \le \lceil \log m \rceil$.
      calculate pza[i,j,h]=
   $\prod_{w=1}^{\lceil \log m \rceil^2}$ prpow[mpa[i,j,h,w],mpb[i,j,h,w]]
      calculate pzb[i,j,h]=
   $\prod_{w=1}^{\lceil \log m \rceil^2}$ ppower[mpa[i,j,h,w],f]$^{mpb[i,j,h,w]}$ mod m
      if pza[i,j,h]=mp[i,j,h] then
          ppower[i,f+1] := pzb[i,j,h]
              spower[j,h,f] mod m.
   5.e all processors $MSI_{i,g,j,h,l}$, $1 \le i, j \le N$,
          $1 \le g, h \le \lceil \log m \rceil$, $1 \le l \le \lceil \log m \rceil^2$.
      msa[i,g,j,h,l] := 1 and msb[i,g,j,h,l] := 1
   5.f all processors $TS_{i,g,j,h}$, $1 \le i, j \le N$,
          $1 \le g, h \le \lceil \log m \rceil$
      ms[i,g,j,h]=spower[i,g,f]*r[j,h] mod m.
   5.g all processors $S_{i,g,j,h,k}$, $1 \le i, j \le N$,
          $1 \le g, h \le \lceil \log m \rceil$, $1 \le k \le B$.
      if prime[k] and $k|ms[i,g,j,h]$ then
          calculate largest e such that
              prpow[k,e]|ms[i,g,j,h].
          choose a random number v
              in the range $1, 2, ..., \lceil \log m \rceil^2$.
          msa[i,g,j,h,v] :=k and msb[i,g,j,h,v] :=e
   5.h all processors $SA_{i,g,j,h}$, $1 \le i, j \le N$, $1 \le g, h \le \lceil \log m \rceil$
      calculate sza[i,g,j,h]=
   $\prod_{w=1}^{\lceil \log m \rceil^2}$ prpow[msa[i,g,j,h,w],msb[i,g,j,h,w]]
      calculate szb[i,g,j,h]=
   $\prod_{w=1}^{\lceil \log m \rceil^2}$ ppower[msa[i,g,j,h,w],f]$^{msb[i,g,j,h,w]}$ mod m
      if sza[i,g,j,h]=mp[i,g,j,h] then
          spower[i,g,f+1] := szb[i,g,j,h]
              spower[j,h,f] mod m.

   5.i all processors $MAI_{j,h,l}$, $1 \le j \le N$,
          $1 \le h \le \lceil \log m \rceil$, $1 \le l \le \lceil \log m \rceil^2 v$.
      maa[j,h,l] := 1 and mab[j,h,l] := 1
   5.j all processors $TA_{j,h}$, $1 \le j \le N$, $1 \le h \le \lceil \log m \rceil$
      ma[j,h]=apower[f]*r[j,h] mod m.
   5.k all processors $A_{j,h,k}$, $1 \le j \le N$,
          $1 \le h \le \lceil \log m \rceil$, $1 \le k \le B$.
      if prime[k] and $k|ma[j,h]$ then
          calculate largest e such that prpow[k,e]|ma[j,h].
          choose a random number v
              in the range $1, 2, ..., \lceil \log m \rceil^2$.
          maa[j,h,v] :=k and mab[j,h,v] :=e
   5.l all processors $AA_{j,h}$, $1 \le j \le N$, $1 \le h \le \lceil \log m \rceil$.
      calculate aza[j,h]=
   $\prod_{w=1}^{\lceil \log m \rceil^2}$ prpow[maa[j,h,w],mab[j,h,w]]
      calculate azb[j,h]=
   $\prod_{w=1}^{\lceil \log m \rceil^2}$ ppower[maa[j,h,w],f]$^{mab[j,h,w]}$ mod m
      if aza[j,h]=ma[j,h] then
          apower[f+1] := azb[j,h]spower[j,h,f] mod m.
```

The action of each processor at each step can be performed by a circuit of polynomial size and polylog depth. A circuit that computes 2^{2^f}POWERS(a,m) for each value of input size n can be obtained by gluing these circuits together. Let α_n be the circuit so obtained for inputs of size n.

To analyse the probability of success, the following notions are introduced. For all $m, N \in \mathcal{Z}_{>0}$, let $\mathcal{C}(m, N)$ be the collection of subsets of $(\mathcal{Z}/m\mathcal{Z})^*$ of size N. For all $T \in \mathcal{C}(m, N)$ and $x \in \mathcal{Z}/m\mathcal{Z}$ define $xT = \{xy \bmod m | y \in T\}$. Say that "$T$ smooths x" if xT contains an $L(m)$-smooth number; call T universal if T smooths x for all $x \in (\mathcal{Z}/m\mathcal{Z})^*$.

Lemma 1 *For all $m \in \mathcal{Z}_{>1}$ with m prime, let $N = L(m)^{c_\bullet} 3 \log m$. Then*

$$\frac{\#\{T \mid T \in \mathcal{C}(m, N) \text{ \& } T \text{ not universal}\}}{\#\mathcal{C}(m, N)} \leq \frac{1}{m^2}$$

Proof: Let m be prime. Let $L = L(m)$ and $N = L(m)^{c_\bullet} 3 \log m$. By theorem 7 the probability that a random positive integer $< m$ is L-smooth is $> 1/L^{c_\bullet}$. Since m is prime the probability that a random element in $(\mathcal{Z}/m\mathcal{Z})^*$ is L-smooth is $> 1/L^{c_\bullet}$. Suppose $T \in \mathcal{C}(m, N)$. Then the probability that T has no L-smooth number is $\leq (1 - 1/L^{c_\bullet})^N$. For all $x \in (\mathcal{Z}/m\mathcal{Z})^*$, x acts as an automorphism on $\mathcal{C}(m, N)$, hence the probability that Tx has no L-smooth number is also $\leq (1 - 1/L^{c_\bullet})^N$. It follows that the probability that T fails to be universal is $\leq m(1 - 1/L^{c_\bullet})^N \leq 1/m^2$ as required.

Lemma 2

1. *For all $n, a, m \in \mathcal{Z}_{>0}$ with m sufficiently large and prime, and $|a| + |m| = n$, α_n on input $< a, m >$ produces an output with probability $> 1/2$.*

2. *For all $n, a, m \in \mathcal{Z}_{>0}$ with m a prime, and $|a| + |m| = n$, if α_n on input $< a, m >$ produces an output, then the output is $a^{2^{2^f}} \bmod m$ for $0 \leq f \leq \lceil \log \log m \rceil$.*

Proof: Consider α_n on inputs a, m. Let N,B be as in the algorithm. Consider the probability that for all h with $0 \leq h \leq \lceil \log m \rceil$ $T_h = \{r[j, h] \mid 1 \leq j \leq N\}$ is universal. By the previous lemma this probability exceed $1/\sqrt{2}$.

Assume the T_h are universal. Fix f such that $0 \leq f \leq \lceil \log \log m \rceil - 1$, and assumme that:

- ppower[i,f], $1 \leq i \leq B$,
- spower[i,h,f], $1 \leq i \leq N$, $1 \leq h \leq \lceil \log m \rceil$, and
- apower[f]

have already been computed.

Consider the set

$$\{ma[j, h] = apower[f] * r[j, h] \mid 1 \leq j \leq N,$$
$$1 \leq h \leq \lceil \log m \rceil\}$$

computed in step 5.j. Since the T_h are universal, this set must contain at least $\lceil \log m \rceil$ B-smooth numbers. If ma[j,h] is B-smooth then a simple analysis shows that with probability $> 1/2$ in step 5.1 $AA_{j,h}$ will write apower[f+1]. It follows that the probability that apower[f+1] is written by some $AA_{j,h}$ exceeds $1 - 1/m$

A similar analysis shows that:

- ppower[i,f+1], $1 \leq i \leq B$,
- spower[i,h,f+1], $1 \leq i \leq N$, $1 \leq h \leq \lceil \log m \rceil$, and
- apower[f+1]

all have probability exceeding $1 - 1/m$ of being written. Let m be sufficiently large. Then $BNh < m$, so with high probability they will all be written. Since $BNh \lceil \log \log m \rceil < m$, with probability $> 1/\sqrt{2}$ they will all be written on every step f.

If apower[f] is written for $0 \leq f \leq \lceil \log \log m \rceil$ then it is easily argued that for all f apower[f]=$a^{2^{2^f}}$ mod m.

Lemma 3 *There exists a $c \in \mathcal{Z}_{>0}$ such that*

1. $D_{exp} = O(\log^3 n)$
where for all $n \in \mathcal{Z}_{>0}$, $D_{exp}(n)$ *=depth of* α_n.

2. $S_{exp} = O(e^{c\sqrt{n \log n}})$
where for all $n \in \mathcal{Z}_{>0}$, $S_{exp}(n)$ *=size of* α_n.

Proof: Using Theorem 2 the result for S_{exp} is straightforward. A naive analysis of depth would give $D_{exp} = O(\log^4 n)$; however, this may be improved to $D_{exp} = O(\log^3 n)$ by using results in [vzG] that the product of n n-bit numbers can be computed by a circuit of depth $\log^2 n$.

Next consider the problem 2^iPOWERS(a,m): when m is prime compute $a^{2^i} \bmod m$, $0 \leq i \leq \lceil \log m \rceil$. This problem is NC^1 reducible to the problem 2^{2^j}POWERS(a,m). A detailed proof is omitted, however the reducing algorithm is as follows:

In the algorithm, a[i] is initialized to -1, indicating not yet computed; otherwise, a[i] holds a^{2^i} mod m. For all $i \in \mathcal{Z}_{>0}$, wt(i) = # of 1's in the binary representation of i.

```
0.   (* Initialize *)
       a[0] = $a$
       for i = 1 to log m in parallel do
           a[i] = -1

1.   (* Compute a[i] for i such that wt(i) = k *)
       for k = 1 to loglog m do
           for i = 0 to log m in parallel do
               if a[i] $\geq$ 0 (* i.e., already computed *)
1.1                compute $2^{2^j}$POWERS(a[i], m)
                   for j = 0 to log log m in parallel do
1.2                    if $2^j$ >i a[i+$2^j$] := $a[i]^{2^{2^j}}$ mod m

2.   (* Output *)
       Output a[i] 0 $\leq$ i $\leq$ log m.
```

Finally, note that it is straightforward to solve MOD-EXP(a, b, m) when m is prime given Algorithm 2^iPOWERS(a,m). First reduce b mod m-1 then compute a^{2^i} for $0 \leq i \leq \lceil \log m \rceil$ using 2^iPOWERS(a,m). Finally, letting $b = \sum_{i=0}^{\lceil \log m \rceil} \beta_i 2^i$ where the $\beta_i = 0$ or 1, compute a^b mod m=$\prod_{i=0,\beta_i=1}^{\lceil \log m \rceil} a^{2^i}$ mod m.

3 GCD

Although a polynomial time sequential algorithm for computing the gcd of two numbers a and b (hereafter denoted (a,b)) has been known for centuries, namely the Euclidean Algorithm, parallelizing the problem has proven very difficult. Algorithms that run in linear and sublinear time have been produced: Brent and Kung show that the gcd of n-bit numbers can be computed in $O(n)$ time with n processors [BK]; Kannan, Miller and Rudolph demonstrate a sub-linear time bound of $n \log \log n / \log n$ using $n^2 \log^2 n$ processors[KMR]; and Chor and Goldreich achieve a time bound of $n/\log n$ using $n^{(1+\epsilon)}$ processors, for any $\epsilon > 0$ [CG]. However, an NC algorithm, if indeed such exists, has not emerged. An algorithm for gcd is presented below which runs in polylog time on a subexponential number of processors.

The algorithm relies on the following principle: let a, b be positive integers, $a \geq b$; let $d = (a, b)$. Then for $r \in \mathcal{Z}$ with $0 \leq r < b$, we have $ra \bmod b = sd$ for some $0 \leq s < b/d$; in fact, for each s, we have exactly d pre-images in the range $0...b-1$. So, as above, by trying sufficiently many random numbers r a smooth s will result. The strategy is thus: pick a random positive integer $r < b$; compute $ra \bmod b$ and divide out the smooth component; if the quotient q divides both a and b, then q the non-smooth part of the gcd. The smooth part of the gcd is obtained by exhaustion.

Let $< a, b >$ be the input of size n, and assume $a \geq b$. Let $B = L(b)$ and $N = L(b)^{c_s}$ where c_s is as in Theorem 7. The following processors are used:

- $PR_{i,j} : 1 \leq i, j \leq B$.
- $SG_i : 1 \leq i \leq B$.
- $LC_j : 1 \leq j \leq N$.
- $SF_{i,j} : 1 \leq i \leq B$, $1 \leq j \leq N$.
- $AF_{i,j,k} : 1 \leq i \leq B$, $1 \leq j \leq N$, $1 \leq k \leq n$.
- $NG_{j,k} : 1 \leq j \leq N$, $1 \leq k \leq n$.
- $SA_v : 1 \leq v \leq n^2$.
- $CA_{i,v} : 1 \leq i \leq B$, $1 \leq v \leq n^2$.

- $CB_{i,v} : 1 \leq i \leq B,\ 1 \leq v \leq n^2$.

In addition, there are the following variables:

- a, b — input.
- prime[i] $1 \leq i \leq B$ — prime[i] is intended to be *true* iff i is prime, initialized to *true* for $i > 1$; prime[1] is initialized to *false*.
- prpow[i, j] $1 \leq i \leq B, 1 \leq j \leq n$, intended to hold i^j.
- r[j] $1 \leq j \leq N$ variables initialized to random numbers from 1,2,...,b-1.
- q[j] $1 \leq j \leq N$ intended to hold coefficient of b in linear combination.
- t[j] $1 \leq j \leq N$ t[j] is intended to hold the j^{th} random linear combination of a and b.
- s[j], u[j] $1 \leq j \leq N$; s[j] u[j] are intended to hold the smooth and non-smooth parts of t[j] respectively.
- ad_1[k], ad_2[j, k] $1 \leq k \leq n^2, 1 \leq j \leq N$; "accumulators" intended to hold prime divisors.
- d_1, d_2 — smooth and non-smooth parts of (a, b).
- a′, a″, b′, b″ — variables related to a and b.
- w — index variable.
- r, q, s — intended to hold coefficients of "winning" linear combination, i.e., $ra' - qb' = s$, s smooth.
- ax[j], ay[j] $1 \leq j \leq N$; intended to hold primes.
- cx[j], cy[j] $1 \leq j \leq N$; intended to hold coefficients for Chinese Remaindering.
- ga, gb — intended to hold numbers relatively prime to b and a respectively.
- bi, binv[k], ai, ainv[k] $1 \leq k \leq n^2$ intended for use in the computation of b^{-1} mod ga and a^{-1} mod gb, respectively.
- r′, r″, q′, q″ — intended for computation of coefficients of extended gcd.

Algorithm GCD(a, b)

```
1. (* Identify all primes $\leq B$ by exhaustion *)
      all processors $PR_{i,j}$ :  $2 \leq i, j \leq B$
         if $j|i$ and $j < i$ then prime[i] := false

2. (* Compute all small powers *)
      all processors $PR_{i,j} : 2 \leq i \leq B, 1 \leq j \leq n$
         compute prpow[i,j] := $i^j$

3. (* Compute smooth factors of $(a, b)$ *)
      all processors $SG_i : 2 \leq i \leq B$
         if prime[i]
            compute largest $e_1$ such that prpow[i,$e_1$]|$a$
            compute largest $e_2$ such that prpow[i,$e_2$]|$b$
            let $e = \min(e_1, e_2)$
            if $e > 0$ then
               v := random integer between 1 and $n^2$
               $ad_1$[v] := prpow[i,e]
               if $ad_1$[v] $\neq$ prpow[i,e] then FAIL

4. (* Compute smooth part of $(a, b)$, replace $a$, $b$ *)
      $d_1 := \prod_{v=1}^{n^2} ad_1[v]$
      $a' := a/d_1$; $b' := b/d_1$

5. (* Find a random linear combination of $a$ and $b$ *)
      all processors $LC_j : 1 \leq j \leq N$
         t[j] := $a' * r[j] \bmod b'$
         q[j] := $\lfloor a' * r[j]/b' \rfloor$

6. (* Find smooth factors *)
      all processors $SF_{i,j} : 1 \leq i \leq B,\ 1 \leq j \leq N$
         if prime[i] and i | t[j]
            v := random integer between 1 and $n^2$
            $ad_2$[j,v] := i

7. (* Compute non-smooth part of gcd *)
      all processors $NG_j : 1 \leq j \leq N$
         s[j] := $\prod_{v=1}^{n^2} ad_2[j,v]$
         u[j] := t[j]/s[j]
         if u[j] | $a$ and u[j] | $b$ then
            $d_2$ := u[j]
            w := j

8. (* Replace a′ and b′; set up linear combination *)
      $a'' := a'/d_2$; $b'' := b'/d_2$
      r := r[w]; q := q[w]; s := s[w,z]
```

```
9. (* Compute prime factors of ga=(a,s[w]) *)
      all processors $SA_v : 1 \le v \le n^2$
         if ad$_2$[w,v]|a$''$ then
            ax[v] := ad$_2$[w,v]
         else ay[v] := ad$_2$[w,v]

10. (* Compute ga=(a,s[w]) *)
      ga := $\prod_{v=1}^{n^2}$ax[v]

11. if ga > 1
    (* Set up b$^{-1}$ mod ga using CRT *)
      all processors $CA_{i,v} : 1 \le i \le B, 1 \le v \le n^2$
         if (ga/ax[v])*i≡1 mod ax[v] then
            cx[v] := (ga/ax[v])*(i mod ax[v])
         if b$''$*i ≡ 1 mod ax[v] then
            binv[v] := i mod ax[v]

12. (* Compute b$^{-1}$ mod ga; divide through by ga *)
      bi := $\sum_{v=1}^{n^2}$binv[v]*cx[v] mod ga
      r$'$ := (r-bi*r*b$''$)/ga; q$'$ := (q-bi*r*a$''$)/ga
      gb := s/ga

13. if gb > 1
    (* Set up a$^{-1}$ mod gb using CRT *)
      all processors $CB_{i,v} : 1 \le i \le B, 1 \le v \le n^2$
         if (gb/ay[v])*i≡1 mod ay[v] then
            cy[v] := (gb/ay[v])*(i mod ay[v])
         if a$''$*i≡1 mod ay[v] then
            ainv[v] := i mod ay[v]

14. (* Compute a$^{-1}$ mod gb; divide through by gb *)
      ai := $\sum_{v=1}^{n^2}$ainv[v]*cy[v] mod gb
      r$''$ := (r$'$-ai*r$'$*b$''$)/sb; q$''$ := (q$'$-ai*r$'$*a$''$)

15. (* Output $(a,b)$, x,y such that ax-by=(a,b) *)
      output <d$_1$*d$_2$, r$''$, q$''$ >
```

It is easily seen that for all $n \in \mathcal{Z}_{>0}$, the action of each processor on inputs of size n can be implemented by a polynomial size circuit of $O(\log^2 n)$ depth. Thus, a circuit for integer gcd can be obtained by gluing the circuits for each processor together. For each n, let the circuit so obtained be called α_n.

Lemma 4 *For all $a, b, n \in \mathcal{Z}_{>0}$, with $|a| + |b| = n$ if α_n terminates on input $< a, b >$, the output is $<(a,b), x, y >$ such that $ax - by = (a,b)$.*

Proof: Let B be as in the algorithm; let $(a,b) = d_1 d_2$ such that $d_1 = \prod p_i^{e_i}$ with p_i prime $\le B$ and $(d_2, p) = 1$ for all primes $p \le B$. If step 3 does not fail, d$_1$is set to d_1. In steps 4–7, $t[j]$ is set to a random linear combination of a' and b', and factored into s[j]u[j], where s[j] is B-smooth. Thus $d_2|$t[j], and $(d_2$,s[j]$) = 1$, so $d_2|$u[j]. If u[j]$|a'$ and u[j]$|b'$, then u[j]$|d_2$, whence u[j]$=d_2$. Thus, if d$_2$is set, it contains d_2. Clearly, w is such that u[w]$=d_2$, hence r and q are such that ra''-qb''=s, with s B-smooth. Also, $(a'', b'') = 1$. Let $(a''$,s$)$ =ga. Then $(b''$,ga$) = 1$. Step 9 partitions the divisors of s into those that divide ga (in ax[.]), and those prime to ga (in ay[.]). Steps 11 and 12 compute bi= b''^{-1}mod sb by the Chinese Remainder Theorem. Then

$$s = ra'' - qb'' = (r - bi * b'' * r)a'' - (q - bi * a'' * r)b''.$$

Since ga$|$(r-bib''r) and ga$|$s, ga$|$(q-bia''r)b''. Since $($ga$, b'') = 1$, ga$|$(q-bia''r). Thus, r$'a''-$q$'b''$=s/ga=gb. Now (sa,a)=1, so a similar computation yields r$''a''-$q$''b'' = 1$. Hence r$''a-$q$''b = (a,b)$.

Analysis of the depth and size of α_n proceeds in a manner similar to that used for modular exponentiation and will not be presented here.

4 Other Results

This section contains a sketch of the proof of Theorem 3,4,5 and 6.

Theorems 4 and 5 can be obtained from straightforward parallelization of any of the well-known algorithms for integer factoring [e.g., Po] and discrete logarithms [Ad] [Po].

Let r be a 'witness' that a, b are relatively prime iff ra mod b is prime and does not divide a or does not divide b. Then an easy argument shows that for a, b relatively prime there are abundant witnesses, but for a, b not relatively prime there are none. Theorem 6.1 follows. Theorem 6.2 follows straightforwardly from the Miller primality test; under ERH, the reduction is deterministic [Mi].

Theorem 3 follows from Theorem 6.2 and Theorem 1.

5 Conclusion

Smoothness has been used to obtain random parallel algorithms for calculating integer gcd and modular exponentiation in polylog time using a subexponential number of processors. Random NC reductions between relative primality, primality and modular exponentiation have been outlined. Finally, random algorithms to perform integer factoring and discrete logarithms in polynomial time with a subexponential number of processors have been indicated.

A number of open problems remain: the obvious ones are to determine the parallel complexity of gcd computation and modular exponentiation. Other questions are:

1. Does there exist evidence that gcd computation (resp., modular exponentiation) is not P-complete? One avenue perhaps is to demonstrate a class co-P (analogous to co-NP), and to show gcd computation lies in P$\bigcap$co-P.
2. Can integer factoring and discrete logarithms be performed in polylog time using a subexponential number of processors?
3. Are the problems of integer gcds, modular exponentiation, and primality testing computable in space($n^{(1/2+o(1))}$) on inputs of size n? It appears that the ideas in this paper may be extendable to yield such results.

References

[Ad] L. M. Adleman, A subexponential algorithm for the discrete log problem, *Proc. 20th Annual Symposium on the Foundations of Computer Science, IEEE* (1979), pp. 55-60.

[AKS] M. Ajtai, J. Komlos and E. Szemeredi, Sorting in c log n parallel steps, *Combinatorica* **3** (1983), pp. 1-19.

[BCH] P. Beame, H. Hoover and S. Cook, Log depth circuits for division and related problems, *SIAM J. Computing* **13** (1986), pp. 994-1003.

[Bo] A. Borodin, Private communication with M. Tompa, August 1985.

[BGH] A. Borodin, J. von zur Gathen and J. Hopcroft, Fast parallel matrix and GCD computation, *Information and Control* **52**, 3 (1982), pp. 241-256.

[BK] R. Brent and H. Kung, A systolic algorithm for integer GCD computation, CMU Tech. Rep. CMU-CS-84-135.

[CG] B. Chor and O. Goldreich, An improved algorithm for integer GCD, MIT Laboratory for Computer Science (1985).

[Co] S. Cook, A Taxonomy of Problems with Fast Parallel Algorithms, *Information and Control* **64** (1985), pp. 2-22.

[Cs] L. Csanky, Fast parallel matrix inversion algorithm, *SIAM J. Comp.,* **5**, 4 (1976), pp. 618-623.

[DKM] C. Dwork, P. Kanellakis and J. Mitchell, On the sequential nature of unification, *J. Logic Programming* (1984), pp. 35-50.

[DLR] D. Dobkin, R. Lipton and S. Reiss, Linear programming is logspace hard for P, *Information Processing Letters* **8** (1979), pp. 96-97.

[Eb] W. Eberly, Very fast matrix and polynomial arithmetic, *Proc. 25th Annual Symposium on Foundations of Computer Science, IEEE* (1984), pp. 21-30.

[FT] F. Fich and M. Tompa, The parallel complexity of exponentiating polynomials over finite fields, Technical Report 85-03-01, University of Washington, Seattle, 1985.

[vzG] J. von zur Gathen, Parallel powering, *Proc. 25th Annual Symposium on Foundations of Computer Science, IEEE* (1984), pp. 31-36.

[GS] M. Goldberg and T. Spencer, A new parallel algorithm for the maximal independent set problem, *Proc. 28th Annual Symposium on Foundations of Computer Science, IEEE* (1987), pp. 161-165.

[GSS] L. Goldschlager, R. Shaw and J. Staples, The maximum flow problem is logspace complete for P, *Theoretical Computer Science* **21** (1982), pp. 105-111.

[KMR] R. Kannan, G. Miller and L. Rudolph, Sublinear parallel algorithm for computing the greatest common divisor of two integers, *SIAM J. Comp.* **16**, 1, (1987), pp. 7-16.

[KW] R. Karp and A. Wigderson, A fast parallel algorithm for the maximal independent set problem, *Proc. 16th Annual ACM Symposium on Theory of Computing* (1984), pp. 266-272.

[La] R. Ladner, The circuit value problem is logspace complete for P, *SIGACT News* **7**, 1 (1975), pp. 18-20.

[Lu] E. Luks, Permutation groups and graph isomorphism, *Proc. 27th Annual Symposium on the Foundations of Computer Science, IEEE* (1986), pp. 292-302.

[Mi] G. Miller, Riemann hypothesis and tests for primality, *J. Computer and System Science* # **13** (1976), pp. 300-317.

[PR1] V. Pan and J. Reif, Efficient parallel solutions of linear systems, *Proc. 17th Annual ACM Symposium on Theory of Computing,* (1985) pp. 143-152.

[PR2] V. Pan and J. Reif, Some polynomial and Toeplitz matrix computations, *Proc. 28th Annual Symposium on Foundations of Computer Science, IEEE* (1987) pp. 173-184.

[Po] C. Pomerance, Fast, rigorous factorization and discrete logarithm algorithms, in *Discrete Algorithms and Complexity*, Academic Press, Florida, 1987.

[Re] J. Reif, Logarithmic depth circuits for algebraic functions, *Proc. 24th Annual Symposium on Foundations of Computer Science, IEEE* (1983), pp. 138-145. Revised version, 1984.

[Ru] W. Ruzzo, On uniform circuit complexity, *J. Computer and System Sciences* **22**, 3 (1981), pp. 365-383.

[Sa] J. Savage, *The Complexity of Computing,* Wiley, New York (1976).