Butler University
Digital Commons @ Butler University

Scholarship and Professional Work - LAS

College of Liberal Arts & Sciences

2010

A Randomized Sublinear Time Parallel GCD Algorithm for the EREW PRAM

Jonathan P. Sorenson
Butler University, jsorenso@butler.edu

Follow this and additional works at: http://digitalcommons.butler.edu/facsch_papers

Part of the Theory and Algorithms Commons

A randomized sublinear time parallel GCD algorithm for the EREW PRAM

Jonathan P. Sorenson

Computer Science and Software Engineering, Butler University, Indianapolis, IN, USA

A R T I C L E   I N F O

A B S T R A C T

We present a randomized parallel algorithm that computes the greatest common divisor of two integers of n bits in length with probability $1 - o(1)$ that takes $O(n \log \log n / \log n)$ time using $O(n^{6+\epsilon})$ processors for any $\epsilon > 0$ on the EREW PRAM parallel model of computation. The algorithm either gives a correct answer or reports failure.

We believe this to be the first randomized sublinear time algorithm on the EREW PRAM for this problem.

1. Introduction

The parallel complexity of computing integer greatest common divisors is an open problem (see [5]), and no new complexity results have been published since the early 1990s. This problem is not known to be either $\mathcal{P}$-complete or in $\mathcal{NC}$ [10,13,16].

The first sublinear time parallel algorithm that uses a polynomial number of processors is due to Kannan, Miller, and Rudolph [12]. Adleman and Kompella [1] presented a randomized algorithm that runs in polylog time, but uses a superpolynomial, yet subexponential number of processors. The fastest currently known algorithm is due to Chor and Goldreich [6] which takes $O(n/\log n)$ time using $O(n^{1+\epsilon})$ processors. See also [22], and Sedjelmaci [19] who showed a clear way to do extended GCDs in the same complexity bounds. However, all of these algorithms use the concurrent-read concurrent-write (CRCW) parallel RAM (PRAM) model of computation.

The algorithms of Chor and Goldreich [6] and the author [22] can be readily modified for the weaker concur-

rent-read exclusive-write (CREW) PRAM to obtain running times of $O(n \log \log n / \log n)$ using a polynomial number of processors. And of course one can take a CRCW PRAM algorithm and emulate it on an exclusive-read exclusive-write (EREW) PRAM at a cost of a factor of $O(\log n)$ in the running time, giving linear time algorithms for the EREW PRAM using a polynomial number of processors. For more information, see [2,7,18,20,21].

In this paper, we present what we believe is the first sublinear time, polynomial processor EREW PRAM algorithm for computing greatest common divisors. Note that the EREW PRAM is weaker than the CREW or CRCW PRAM models of parallel computation. We do make use of random numbers in a fundamental way.

Theorem 1.1. *There exists a randomized algorithm to compute the greatest common divisor of two integers of total length n in binary with probability $1 - o(1)$ in $O(n \log \log n / \log n)$ time using a polynomial number of processors on the EREW PRAM.*

In the next section we describe our algorithm, and in Section 3 we prove correctness, give a complexity analysis, and flesh out the details of the algorithm. We conclude in Section 4 with a simple result on the relative density of

integers with large polynomially smooth divisors, which is needed for the analysis of the algorithm.

2. Algorithm description

Define the inputs as u, v of total length n in binary. Let B, our small prime bound, be defined as $B = B(n) := n^2$. A larger value for B can be chosen, so long as $\log B = o(n)$, but correctness would be compromised if B were significantly smaller (see Section 3.1).

1. Find a list of primes up to B. Also, for each prime $p \leqslant B$, compute and save p^e for $e = 1 \ldots \lfloor n/\log_2 p \rfloor$.
2. Remove and save common prime factors of u, v that are $\leqslant B$, and let u_0, v_0 denote these modified inputs. WLOG we assume $u_0 \geqslant v_0$.
3. **Main Loop**. Repeat while $u_i v_i \neq 0$. Here i indicates the current loop iteration, starting at $i = 0$.
    (a) For $j := 1$ to $2B \log n$ in parallel do:
        i. Choose a_{ij} uniformly at random from $1 \ldots v_i - 1$.
        ii. Compute $r_{ij} := a_{ij} u_i \bmod v_i$.
        iii. Compute s_{ij} as r_{ij} with all prime factors $p \leqslant B$ removed.
        (We elaborate on how to do this below.)
    (b) Find $s_i := \min_j \{s_{ij}\}$. Let $j_{\min}$ denote the value of j for which $s_i = s_{ij}$, and for later reference, let $a_i = a_{ij_{\min}}$.
    (c) $u_{i+1} := v_i$; $v_{i+1} := s_i$.
4. $u_i + v_i$ is, with probability $1 - o(1)$, equal to $\gcd(u_0, v_0)$ (as we show below). If we err, it is by including spurious factors that do not belong, so verify that $u_i + v_i$ evenly divides both u_0, v_0, and if not, report an error. Otherwise, include any saved common prime factors found in step 2 above, and the algorithm is complete.

3. Algorithm analysis

In this section we prove correctness, and compute the parallel complexity of our algorithm from the previous section.

3.1. Correctness

Note that in Steps 2 and 4 we handle any prime divisors $\leqslant B$ of the $\gcd(u, v)$, so WLOG we can assume either $\gcd(u, v) = 1$ or $\gcd(u, v) > B$.

At iteration i of the main loop, we perform the transformation

$$(u_i, v_i) \rightarrow (v_i, s_i).$$

Since s_i is equal to $a_i u_i \bmod v_i$, ignoring factors below B, this transformation will only fail to preserve the greatest common divisor if a_i and v_i share a common factor. Furthermore, this common factor must be composed only of primes exceeding B. Since a_i is chosen uniformly at random, the probability a_i and v_i share a prime factor larger that B is at most

$$\sum_{p \mid v_i, \, p > B} \frac{1}{p} \leqslant \sum_{p \mid v_i, \, p > B} \frac{1}{B} \leqslant \frac{\log_B v_i}{B} = O\left(\frac{1}{n \log n}\right).$$

As we will see below, with high probability, the number of main loop iterations is $o(n)$. Thus, the probability that any of the a_i values introduces a spurious factor is $o(1)$.

Note that in [23], a similar, but not identical, transformation was analyzed. It was observed that in practice, with *no* removal of small prime divisors, the expected number of bits contributed by spurious factors was constant per main loop iteration.

3.2. Runtime analysis

First we calculate the number of main loop iterations, and then we describe how each iteration can be computed in $O(\log n)$ time using a polynomial number of processors.

3.2.1. Main loop iterations

Let $W := 0.5(\log B)^2/\log\log B$. Then by Theorem 4.2, which we prove in the next section, the length of s_{ij} is smaller than $r_{ij} < v_i$ by at least $\Theta(W)$ bits with probability at least $1/B$. (Note that we chose 0.5 to get a clean $1/B$ probability – other choices for the constant can be made to work with the right adjustments.)

So, the probability all $2B \log n$ choices for j fail to have $\log s_{ij} \leqslant \log v_i - W$ is

$$\left(1 - \frac{1}{B}\right)^{2B \log n} = O\left(\frac{1}{n^2}\right).$$

So, with probability $1 - O(1/n^2)$, $\log s_i \leqslant \log v_i - W$.

We remove roughly $(\log B)^2/\log\log B$ bits each main loop iteration. Thus, the number of main loop iterations is $O(n \log\log B/(\log B)^2) = o(n)$. The probability that *any one* loop iteration fails to remove the needed $\Theta(W)$ bits is $O(1/n)$, so the probability we exceed this number of main loop iterations and terminate without computing an answer is $o(1)$.

3.2.2. Computation cost and algorithm details

Unless stated otherwise, cost is given for the EREW PRAM. For a brief overview of the cost of parallel arithmetic, see [22, Section 6.2].

Step 1. We can find the primes $\leqslant B$ in $O(\log B)$ time using $O(B)$ processors (see [24]). For each prime $p \leqslant B$ and $e \leqslant n$, we can compute p^e in at most $O(\log n)$ multiplications, each of which takes $O(\log B)$ time using $B^{1+o(1)}$ processors [17]. See also [16, Theorem 12.2]. As there are $O(B/\log B)$ primes, this is $O(\log n \log B)$ time using $nB^{2+o(1)}$ processors.

Step 2. For each prime p and exponent e, we assign a group of processors to see if p^e divides u but p^{e+1} does not. Division takes $O(\log n)$ time using $n^{1+\epsilon}$ processors for any $\epsilon > 0$ using the algorithm of Beame, Cook, and Hoover [3], giving a total processor count of $O(n^{2+\epsilon} B/\log B)$.

The result is a vector of the form $[p_k^{e_k}]$ that lists the primes dividing u with maximal exponents. Since there are at most $n/\log B$ integers in the vector > 1, and they total at most n bits (their product is $\leqslant u$), the iterated product algorithm of [3] can take their product in $O(\log n)$ time using $n^{1+\epsilon}$ processors. Dividing u by this product can be done at the same cost.

We repeat this for v, and obtain a similar vector.

We combine these two vectors using a minimum operation, and take the product of the entries, to obtain the shared prime power divisors of u, v which must be saved for Step 4.

The total cost of this step is $O(\log n)$ time using $O(n^{2+\epsilon} B/\log B)$ processors.

See also [8] and references therein.

Step 3. Checking for zero takes $O(\log n)$ time using $O(n)$ processors.

Step 3(a)i. For each j, choosing an n-bit number at random takes constant time using $O(n)$ processors. We reduce it modulo v_i in $O(\log n)$ time using $n^{1+\epsilon}$ processors [3].

Step 3(a)ii. This is simply a multiplication and a division, again taking $O(\log n)$ time using $n^{1+\epsilon}$ processors.

Step 3(a)iii. Here we use the same method as described in Step 2 above. This is $O(\log n)$ time using $O(n^{2+\epsilon} B/\log B)$ processors for each j.

Step 3(a). And so, the total cost of this parallel step is $O(\log B)$ time using $O(n^{2+\epsilon}(\log n)B^2/\log B)$ processors.

Step 3(b). This can be done in $O(\log(B \log n)) = O(\log B)$ time using $O(B \log n)$ processors.

Step 3(c). This takes constant time using $O(n)$ processors.

We conclude that the cost of one main loop iteration is $O(\log B)$ time using $O(n^{2+\epsilon}(\log n)B^2/\log B)$ processors. Step 3(a)iii is the bottleneck.

Earlier we showed that the number of iterations is $O(n \log \log B/(\log B)^2)$, for a total time of $O(n \log \log B/\log B)$ for all iterations of the main loop.

Step 4. This is an addition, a division, and a multiplication using the results from Step 2; $O(\log n)$ time using $n^{1+\epsilon}$ processors.

Clearly, the bottleneck of the algorithm is Step 3(a). The overall complexity is

$$O\left(\frac{n \log \log B}{\log B}\right) = O\left(\frac{n \log \log n}{\log n}\right) \quad \text{time, and}$$

$$O\left(n^{2+\epsilon}(\log n)\frac{B^2}{\log B}\right) = O(n^{6+\epsilon}) \quad \text{processors,}$$

where $\epsilon > 0$,

for the EREW PRAM. This completes our proof of Theorem 1.1.

One could take B to be superpolynomial in n; for example, if $B = \exp[\sqrt{n}]$ we can obtain a running time of roughly $\sqrt{n}$ using $\exp[O(\sqrt{n})]$ processors. Similar results could be obtained from some of the CRCW PRAM algorithms mentioned in the introduction by porting them to the EREW PRAM and setting parameters appropriately.

We can also obtain an $O(n/\log n)$ running time on the randomized CRCW PRAM; see [4] for how to perform the necessary main loop operations in $O(\log n/\log \log n)$ time via the explicit Chinese remainder theorem. See also [8].

It would be interesting to see if this algorithm can be modified to compute Jacobi symbols quickly in parallel. See [9] and references therein.

4. Numbers with smooth divisors

Let $P(n)$ denote the largest prime divisor of n. If $P(n) \leqslant y$ we say that n is y-smooth. Let

$$\Psi(x, y) = \#\{n \leqslant x: P(n) \leqslant y\},$$

the number of integers $\leqslant x$ that are y-smooth. Let $u = u(x, y) := \log x/\log y$. We will make use of the following lemma.

Lemma 4.1. *(See [11, Corollary 1.3].) Let $\epsilon > 0$ and assume $u < y^{1-\epsilon}$. Then*

$$\Psi(x, y) = xu^{-u(1+o(1))} \quad \text{for } x > y \geqslant 2.$$

Note that the $o(1)$ here tends to zero for large u, and the implied constant depends on ϵ. Better results are known, but this suffices for our purposes. For additional references see [11,25], and for references on approximation algorithms for $\Psi(x, y)$ see [14].

We recall the definition of H_k, the kth harmonic number as

$$H_k = \sum_{i=1}^{k} \frac{1}{i}.$$

It is well known that $H_k = \log k + \gamma + O(1/k)$, where $\gamma = 0.57721\ldots$ is Euler's constant (for example, see [15, 4.5.4]).

Fix a constant $c > 0$. Define $B(x)$ to be a strictly increasing function of x, but with $\log B(x) = o(\log x)$. (We are primarily interested in $B(x)$ polynomial in $\log x$.) Define

$$W(x) := \frac{c \cdot (\log B(x))^2}{\log \log B(x)},$$

$$F(x) := \#\{n \leqslant x: n = my, \ P(m) \leqslant B(x), \ \log m \geqslant W(x)\}.$$

In other words, $F(x)$ counts integers $n \leqslant x$ where n has a $B(x)$-smooth divisor that is $\geqslant \exp W(x)$.

Theorem 4.2. *Let $\epsilon > 0$. For sufficiently large x we have*

$$F(x) \geqslant \frac{x}{B(x)^{c(1+\epsilon)}}.$$

Proof. Choose $\delta > 0$ such that $(1 + \delta)^3 < 1 + \epsilon$. From the definition, we have

$$F(x) = \sum_{y=1}^{x/\exp[W(x)]} \Psi\left(\frac{x}{y}, B(x)\right).$$

First, we limit the range of summation to obtain the lower bound

$$F(x) \geqslant \sum_{y=x/(\exp[(1+\delta)W(x)])}^{x/\exp[W(x)]} \Psi\left(\frac{x}{y}, B(x)\right).$$

Next, we apply Lemma 4.1. We also observe that $u^{-u(1+o(1))} \geqslant u^{-(1+\delta)u}$ for large u, and for a lower bound, we can fix u at its largest value on the interval of summation, namely $u = u(x) = (1 + \delta)W(x)/\log B(x)$. This gives us

$$F(x) \geqslant \sum_{y=x/(\exp[(1+\delta)W(x)])}^{x/\exp[W(x)]} \frac{x}{y} \cdot u^{-(1+\delta)u}.$$

Using $\sum_{a}^{b} 1/t = H_b - H_a \geqslant (1-\delta)\log(b/a)$ for sufficiently large a, we obtain that

$$F(x) \geqslant x \cdot \delta(1-\delta)W(x) \cdot u^{-(1+\delta)u}$$
$$\geqslant x \cdot u^{-(1+\delta)u}$$

as W is a strictly increasing function of x for large x. Next we plug in for u as follows:

$$\log\left(u^{-(1+\delta)u}\right)$$
$$= -(1+\delta)u \log u$$
$$= -(1+\delta)\frac{(1+\delta)W(x)}{\log B(x)} \log\left(\frac{(1+\delta)W(x)}{\log B(x)}\right)$$
$$= -(1+\delta)\frac{(1+\delta)\,c\,\log B(x)}{\log\log B(x)} \log\left(\frac{(1+\delta)\,c\,\log B(x)}{\log\log B(x)}\right)$$
$$\geqslant -c\,(1+\delta)^3 \log B(x)$$

for x sufficiently large. We now have

$$F(x) \geqslant x \cdot B(x)^{-c(1+\delta)^3}$$

for sufficiently large x. $\quad\square$

Only a lower bound is needed for our purposes, but one can obtain an upper bound on $F(x)$ of similar shape using the same general methods.

Acknowledgements

References

[1] L.M. Adleman, K. Kompella, Using smoothness to achieve parallelism, in: 20th Annual ACM Symposium on Theory of Computing, 1988, pp. 528–538.

[2] Eric Bach, Jeffrey O. Shallit, Algorithmic Number Theory, vol. 1, MIT Press, 1996.

[3] P.W. Beame, S.A. Cook, H.J. Hoover, Log depth circuits for division and related problems, SIAM J. Comput. 15 (1986) 994–1003.

[4] Daniel J. Bernstein, Jonathan P. Sorenson, Modular exponentiation via the explicit Chinese remainder theorem, Math. Comp. 76 (257) (2007) 443–454.

[5] A. Borodin, J. von zur Gathen, J. Hopcroft, Fast parallel matrix and GCD computations, Inform. and Control 52 (1982) 241–256.

[6] B. Chor, O. Goldreich, An improved parallel algorithm for integer GCD, Algorithmica 5 (1990) 1–10.

[7] R. Crandall, C. Pomerance, Prime Numbers, a Computational Perspective, Springer, 2001.

[8] George Davida, Bruce Litow, Guangwu Xu, Fast arithmetics using Chinese remaindering, Inform. Process. Lett. 109 (13) (2009) 660–662.

[9] Shawna M. Meyer Eikenberry, Jonathan P. Sorenson, Efficient algorithms for computing the Jacobi symbol, J. Symbolic Comput. 26 (4) (1998) 509–523.

[10] R. Greenlaw, H.J. Hoover, W.L. Ruzzo, Limits to Parallel Computation, Oxford University Press, 1995.

[11] A. Hildebrand, G. Tenenbaum, Integers without large prime factors, J. Théor. Nombres Bordeaux 5 (1993) 411–484.

[12] R. Kannan, G. Miller, L. Rudolph, Sublinear parallel algorithm for computing the greatest common divisor of two integers, SIAM J. Comput. 16 (1) (1987) 7–16.

[13] R. Karp, V. Ramachandran, Parallel algorithms for shared-memory machines, in: J. van Leeuwen (Ed.), Algorithms and Complexity, in: Handbook of Theoretical Computer Science, vol. A, Elsevier and MIT Press, 1990.

[14] Scott Parsell, Jonathan P. Sorenson, Fast bounds on the distribution of smooth numbers, in: Florian Hess, Sebastian Pauli, Michael Pohst (Eds.), Proceedings of the 7th International Symposium on Algorithmic Number Theory (ANTS-VII), in: LNCS, vol. 4076, Springer, Berlin, Germany, ISBN 3-540-36075-1, July 2006, pp. 168–181.

[15] P. Purdom Jr., C. Brown, The Analysis of Algorithms, Holt, Rinehart, and Winston, 1985.

[16] J.H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufman, San Mateo, CA, 1993.

[17] A. Schönhage, V. Strassen, Schnelle Multiplikation großer Zahlen, Computing (Arch. Elektron. Rechnen) 7 (1971) 281–292, MR 45 #1431.

[18] Sidi Mohamed Sedjelmaci, On a parallel extended euclidean algorithm, in: AICCSA, IEEE Computer Society, 2001, pp. 235–241.

[19] Sidi Mohamed Sedjelmaci, A parallel extended GCD algorithm, J. Discrete Algorithms 6 (3) (2008) 526–538.

[20] Sidi Mohamed Sedjelmaci, On a parallel Lehmer–Euclid GCD algorithm, in: Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation, ACM, New York, 2001, pp. 303–308.

[21] Sidi Mohamed Sedjelmaci, A modular reduction for GCD computation, J. Comput. Appl. Math. 162 (1) (2004) 17–31.

[22] Jonathan P. Sorenson, Two fast GCD algorithms, J. Algorithms 16 (1994) 110–144.

[23] Jonathan P. Sorenson, An analysis of the generalized binary gcd algorithm, in: Alf van der Poorten, Andreas Stein (Eds.), High Primes and Misdemeanors: Lectures in Honour of the 60th Birthday of Hugh Cowie Williams, Banff, Alberta, Canada, 2004, pp. 327–340.

[24] Jonathan P. Sorenson, Ian Parberry, Two fast parallel prime number sieves, Inform. and Comput. 144 (1) (1994) 115–130.

[25] Gérald Tenenbaum, Introduction to Analytic and Probabilistic Number Theory, English edition, Cambridge Stud. Adv. Math., vol. 46, Cambridge University Press, 1995.