MODULAR EXPONENTIATION VIA THE EXPLICIT CHINESE REMAINDER THEOREM

DANIEL J. BERNSTEIN AND JONATHAN P. SORENSON

ABSTRACT. Fix pairwise coprime positive integers $p_1, p_2, \ldots, p_s$. We propose representing integers u modulo m, where m is any positive integer up to roughly $\sqrt{p_1 p_2 \cdots p_s}$, as vectors $(u \bmod p_1, u \bmod p_2, \ldots, u \bmod p_s)$. We use this representation to obtain a new result on the parallel complexity of modular exponentiation: there is an algorithm for the Common CRCW PRAM that, given positive integers x, e, and m in binary, of total bit length n, computes $x^e \bmod m$ in time $O(n/\lg\lg n)$ using $n^{O(1)}$ processors.

1. INTRODUCTION

In this paper we consider the problem of computing $x^e \bmod m$ for large integers x, e, and m. This is the bottleneck in Rabin's algorithm for testing primality, the Diffie-Hellman algorithm for exchanging cryptographic keys, and many other common algorithms. See, e.g., [19, Section 4.5.4].

The usual solution is to compute small integers that are congruent modulo m to various powers of x. See, e.g., [19, Section 4.6.3], [9, Section 1.2], [14], [21], and [7]. For example, say $e = 10$. One can compute $x_1 = x \bmod m$, then $x_2 = x_1^2 \bmod m$, then $x_4 = x_2^2 \bmod m$, then $x_5 = x_1 x_4 \bmod m$, and finally $x_{10} = x_5^2 \bmod m$. It is often convenient to allow x_1, x_2, x_4, x_5 to be slightly larger than m.

The output $x^e \bmod m$ and inputs x, e, m are conventionally written in binary. Standard practice is to also use the binary representation for x_1, x_1^2, x_2, x_2^2, etc. We instead use the **residue representation**: x_1 is represented as $(x_1 \bmod p_1, x_1 \bmod p_2, \ldots, x_1 \bmod p_s)$, x_1^2 is represented as $(x_1^2 \bmod p_1, x_1^2 \bmod p_2, \ldots, x_1^2 \bmod p_s)$, etc. Here $p_1, p_2, \ldots, p_s$ are small pairwise coprime positive integers, typically primes, such that $P = p_1 p_2 \cdots p_s$ is sufficiently large. Note that, because p_j is small, it is easy to compute $x_1^2 \bmod p_j$ from $x_1 \bmod p_j$.

The usual Chinese remainder theorem says that $x_1^2 \bmod P$ is determined by the residue representation of x_1^2. In fact, any integer u is congruent modulo P to a particular linear combination of $u \bmod p_1, u \bmod p_2, \ldots, u \bmod p_s$. The *explicit* Chinese remainder theorem says, in a computationally useful form, exactly what multiple of P must be subtracted from the linear combination to obtain u. See Section 2.

Once we know the binary representation of x_1^2, we could reduce it modulo m to obtain the binary representation of x_2. We actually use the explicit CRT modulo

---

Date: 2003.08.15. Permanent ID of this document: `42983bacd3ca91e10217a79c66508b3b`.

2000 *Mathematics Subject Classification.* Primary 11Y16; Secondary 68W10.

This paper supersedes the preliminary papers [5] by Bernstein and [31] by Sorenson. Bernstein was supported by the National Science Foundation under grants DMS-9600083 and DMS–9970409. Sorenson was supported by the National Science Foundation under grant CCR–9626877. Sorenson completed part of this work while on sabbatical at Purdue University in Fall 1998.

$$
\begin{tikzcd}
\text{Residue representation of } x_1 \arrow[d, "\text{easy}"] \\
\text{Residue representation of } x_1^2 \arrow[d, "\text{CRT}"] \arrow[dd, bend right=60, "\text{ECRT}"'] \arrow[ddd, bend right=70, "\text{ECRT mod } m"'] \arrow[dddd, bend right=80, "\text{ECRT mod } m \text{ mod } p_j"'] \\
\text{Binary representation of something congruent to } x_1^2 \text{ modulo } P \arrow[d, "\text{reduction modulo } P"] \\
\text{Binary representation of } x_1^2 \arrow[d, "\text{reduction modulo } m"] \\
\text{Binary representation of } x_2 \arrow[d, "\text{reduction modulo } p_j"] \\
\text{Residue representation of } x_2
\end{tikzcd}
$$

m to obtain the binary representation of x_2 directly from the residue representation of x_1^2. See Section 3.

Once we know the binary representation of x_2, we reduce it modulo each p_j to obtain the residue representation of x_2. An alternative is to use the explicit CRT modulo m modulo p_j to obtain the residue representation of x_2 directly from the residue representation of x_1^2. See Section 4.

If P is sufficiently large then, as discussed in Section 5, one can perform several multiplications in the residue representation before reduction modulo m. This is particularly beneficial for parallel computation: operations in the residue representation are more easily parallelized than reductions modulo m. By optimizing parameters, we obtain the following result:

Theorem 1.1. *There is an algorithm for the Common CRCW PRAM that, given the binary representations of positive integers x, e, and m, of total bit length n, computes the binary representation of $x^e \bmod m$ in time $O(n/\lg\lg n)$ using $n^{O(1)}$ processors.*

Section 6 defines the Common CRCW PRAM. Section 8 presents the algorithm. Section 7 presents a simpler algorithm taking time $O(n)$.

The same techniques can easily be applied to exponentiation in finite rings more general than $\mathbf{Z}/m$. For example, the deterministic polynomial-time primality test of Agrawal, Kayal, and Saxena in [2] can be carried out in sublinear time using a polynomial number of processors on the Common CRCW PRAM; the bottleneck is exponentiation in a ring of the form $(\mathbf{Z}/m)[x]/(x^k - 1)$.

2. The explicit Chinese remainder theorem

A well-known method to multiply integers in the binary representation is to convert them to the residue representation, multiply them in the residue representation, and convert the product to the binary representation. This idea was introduced in the 1950s by Svoboda, Valach, and Garner, according to [19, Section 4.3.2]. This section discusses conversion from the residue representation back to the binary representation.

For each real number α such that $\alpha - 1/2 \notin \mathbf{Z}$, define $\operatorname{round}\alpha$ as the unique integer r with $|r-\alpha| < 1/2$.

Theorem 2.1. *Let $p_1, p_2, \ldots, p_s$ be pairwise coprime positive integers. Write $P = p_1 p_2 \cdots p_s$. Let $q_1, q_2, \ldots, q_s$ be integers with $q_i P/p_i \equiv 1 \pmod{p_i}$. Let u be an integer with $|u| < P/2$. Let $t_1, t_2, \ldots, t_s$ be integers with $t_i \equiv uq_i \pmod{p_i}$. Then $u = P\alpha - P\operatorname{round}\alpha$ where $\alpha = \sum_i t_i/p_i$.*

Proof. $P\alpha = \sum_i t_i(P/p_i) \equiv uq_i(P/p_i) \equiv u \pmod{p_i}$ for each i, so $P\alpha \equiv u \pmod{P}$. Write $r = \alpha - u/P$. Then r is an integer, and $|r-\alpha| = |u/P| < 1/2$, so $r = \operatorname{round}\alpha$, i.e., $u/P = \alpha - \operatorname{round}\alpha$. □

The usual Chinese remainder theorem says that $u \equiv \sum_i t_i P/p_i \pmod{P}$; this is the integer version of Lagrange's interpolation formula. This is a popular way to compute u from the residues $u \bmod p_i$: first compute $t_i = ((u \bmod p_i)q_i) \bmod p_i$ or simply $t_i = (u \bmod p_i)q_i$, then compute $P\alpha = \sum_i t_i P/p_i$, then reduce $P\alpha$ modulo P to the right range.

The explicit Chinese remainder theorem, Theorem 2.1, suggests another way to divide $P\alpha$ by P. Use t_i and p_i directly to compute a low-precision approximation to $\alpha = \sum_i t_i/p_i$ with sufficient accuracy to determine $\operatorname{round}\alpha$; see, for example, Theorem 2.2 below. Then subtract $P\operatorname{round}\alpha$ from $P\alpha$ to obtain u.

As far as we know, the first use of the explicit Chinese remainder theorem was by Montgomery and Silverman in [24, Section 4]. It has also appeared in [23], [11, Section 2.1], [5], [31], and [1, Section 5].

Theorem 2.2. *Let $\beta_1, \beta_2, \ldots, \beta_s$ be real numbers. Let r and a be integers. If $|r - \sum_i \beta_i| < 1/4$ and $2^a \ge 2s$ then $r = \lfloor 3/4 + 2^{-a}\sum_i \lfloor 2^a\beta_i\rfloor\rfloor$.*

Proof. $r < 1/4 + \sum_i \beta_i = 1/4 + 2^{-a}\sum_i 2^a\beta_i \le 1/4 + 2^{-a}(s + \sum_i \lfloor 2^a\beta_i\rfloor) \le 1/4 + 2^{-a}(2^a/2 + \sum_i \lfloor 2^a\beta_i\rfloor) = 3/4 + 2^{-a}\sum_i \lfloor 2^a\beta_i\rfloor \le 3/4 + \sum_i \beta_i < r+1$. □

In the situation of Theorem 2.1, assume further that $|u| < P/4$. We can then use Theorem 2.2 to quickly compute $r = \operatorname{round}\alpha$. We choose a with $2^a \ge 2s$, then compute the fixed-point approximations $2^{-a}\lfloor 2^a t_i/p_i\rfloor$ to t_i/p_i, then compute $r = \lfloor 3/4 + 2^{-a}\sum_i \lfloor 2^a t_i/p_i\rfloor\rfloor$.

3. The explicit CRT mod m

Theorem 3.1. *Let $p_1, p_2, \ldots, p_s$ be pairwise coprime positive integers. Write $P = p_1 p_2 \cdots p_s$. Let $q_1, q_2, \ldots, q_s$ be integers with $q_i P/p_i \equiv 1 \pmod{p_i}$. Let u be an integer with $|u| < P/2$. Write $t_i = uq_i \bmod p_i$. Let m be a positive integer. Write $v = \sum_i t_i(P/p_i \bmod m) - (P \bmod m)\operatorname{round}\sum_i t_i/p_i$. Then $u \equiv v \pmod{m}$, and $|v| \le m\sum_i p_i$.*

The hypotheses on p, P, q, u are the same as in Theorem 2.1. The hypothesis on t is more restrictive: Theorem 2.1 allowed any integer $t_i \equiv uq_i \pmod{p_i}$, whereas Theorem 3.1 insists that $0 \le t_i < p_i$.

One can allow a slightly wider range of t_i, often making t_i easier to compute, at the expense of a larger bound on $|v|$. In the other direction, one can reduce the bound on $|v|$ by taking t_i between $-p_i/2$ and $p_i/2$, and one can reduce the bound further by similarly adjusting $P/p_i \bmod m, P \bmod m$.

Proof. By Theorem 2.1, $u = \sum_i t_i P/p_i - P \operatorname{round} \sum_i t_i/p_i \equiv v$. Furthermore, $v \le \sum_i t_i(P/p_i \bmod m) \le \sum_i p_i m$ since $t_i \le p_i$, and $-v \le (P \bmod m) \operatorname{round} \sum_i t_i/p_i \le m \sum_i p_i$ since $t_i \le p_i^2$. □

Theorem 3.1 suggests the following representation of integers modulo m. Select moduli $p_1, \ldots, p_s$ whose product $P = p_1 \cdots p_s$ exceeds $4(m \sum_i p_i)^2$. Use the vector $(x \bmod p_1, \ldots, x \bmod p_s)$, where x is any integer between $-m \sum_i p_i$ and $m \sum_i p_i$, to represent $x \bmod m$. Note that each element of $\mathbf{Z}/m$ has many representations.

The following procedure, given two such vectors $(x \bmod p_1, \ldots, x \bmod p_s)$ and $(y \bmod p_1, \ldots, y \bmod p_s)$, computes another such vector $(v \bmod p_1, \ldots, v \bmod p_s)$ with $v \equiv xy \pmod{m}$:

- Precompute q_i, $P/p_i \bmod m$, and $P \bmod m$.
- Compute $t_i = ((x \bmod p_i)(y \bmod p_i)q_i) \bmod p_i$, so that $t_i = (xy)q_i \bmod p_i$.
- Compute $\operatorname{round} \sum_i t_i/p_i$ by Theorem 2.2.
- Compute $v = \sum_i t_i(P/p_i \bmod m) - (P \bmod m) \operatorname{round} \sum_i t_i/p_i$. Then v is between $-m \sum_i p_i$ and $m \sum_i p_i$, and $v \equiv xy \pmod{m}$, by Theorem 3.1.
- Compute the residues $(v \bmod p_1, \ldots, v \bmod p_s)$.

The output can then be used in subsequent multiplications. One can carry out more componentwise operations before applying Theorem 3.1 if P is chosen larger; see Section 5 for further discussion.

Montgomery and Silverman in [24, Section 4] suggested computing $u \bmod m$ by first computing v. (Another way to compute $u \bmod m$, well suited for FFT-based arithmetic, is to perform the computation of [6, Section 13] modulo m.) The idea of using v for subsequent operations, and not bothering to compute $u \bmod m$, was introduced in [5].

4. The explicit CRT mod m mod p_j

Theorem 4.1. *Under the assumptions of Theorem 3.1,*

$$v \equiv \sum_i t_i(P/p_i \bmod m \bmod p_j) - (P \bmod m \bmod p_j) \operatorname{round} \sum_i t_i/p_i \pmod{p_j}.$$

Proof. Reduce the definition of v modulo p_j. □

Section 3 discussed precomputing $P/p_i \bmod m$, precomputing $P \bmod m$, computing v, and reducing v modulo p_j. Theorem 4.1 suggests a different approach: precompute $P/p_i \bmod m \bmod p_j$; precompute $P \bmod m \bmod p_j$; compute $v \bmod p_j$ as $(\sum_i t_i(P/p_i \bmod m \bmod p_j) - (P \bmod m \bmod p_j) \operatorname{round} \sum_i t_i/p_i) \bmod p_j$. This idea was introduced in [5].

Theorem 4.1 is particularly well suited to parallel computation, as briefly observed in [5]. By using additional ideas described in the remaining sections of this paper, one can exponentiate on a polynomial-size parallel computer in sublinear time, as shown in [31]. This paper supersedes [5] and [31].

5. Higher powers

We begin by reviewing the left-to-right base-2 algorithm for computing $x^e \bmod m$:

Let l denote the number of bits in e;
Write $e = \sum_{k=0}^{l-1} e_k 2^k$, where $e_k \in \{0, 1\}$;
$y \leftarrow 1$;
For($k \leftarrow l - 1$; $k \geq 0$; $k \leftarrow k - 1$) do:
    $y \leftarrow y^2 x^{e_k} \bmod m$;
Output(y);

More generally, let b be a power of 2. The base-b algorithm is as follows:

Let l denote the number of base-b digits in e;
Write $e = \sum_{k=0}^{l-1} e_k b^k$, where $0 \leq e_k < b$;
$y \leftarrow 1$;
For($k \leftarrow l - 1$; $k \geq 0$; $k \leftarrow k - 1$) do:
    $y \leftarrow y^b x^{e_k} \bmod m$;
Output(y);

Sane people take every opportunity to reduce intermediate results modulo m: for example, they compute $y^b \bmod m$ by squaring y, reducing modulo m, squaring again, reducing again, etc. We call this quite reasonable practice into question. It may be better to first compute $y^b x^{e_k}$, then reduce the result modulo m. The benefit of performing fewer reductions may outweigh the cost of performing arithmetic on larger numbers. (Similar comments apply to other exponentiation algorithms.)

When we use the explicit Chinese remainder theorem to perform multiplications modulo m, the cost of handling larger numbers is in using more primes $p_1, p_2, \ldots$. In the context of parallel computation, increasing the number of primes mainly affects the number of processors, not the run time. In fact, as explained in the next three sections, we can save an asymptotically non-constant factor in the run time in various models of parallel computation, at a reasonable expense in the number of processors.

Perhaps the same idea can also save a small constant factor in the serial case.

6. Definition of the CREW PRAM and the Common CRCW PRAM

We use two models of parallel computation in this paper. This section defines the models.

In both models, computers are "parallel random-access machines" (PRAMs) in which many processors can access variable locations in a shared memory. The models differ in how they handle memory conflicts: the "Common CRCW PRAM" allows many processors to simultaneously write a common value to a single memory location, while the "CREW PRAM" does not. Our exponentiation algorithm in Section 8 relies on the extra power of the Common CRCW PRAM; the simplified algorithm in Section 7 works in either model.

We caution the reader that phrases such as "CREW PRAM" are not sufficient to pin down a model of computation in the literature. For example, [26, Sections 2.6 and 15.2] defines unrealistic PRAMs in which the processor word size is permitted to grow linearly as a computation progresses; at the other extreme, [12] presents computations that fit into logarithmic-size index registers and constant-size data registers. The complexity of a computation can depend heavily on details of the model. See [13] for a comparison of several models.

Computers. A parallel computer is parametrized by positive integers p, s, w, r, i such that $p \leq 2^w$ and $s \leq 2^w$. The computer has p processors, numbered from 0 through $p-1$, operating on s words of shared memory, labelled $\texttt{mem}[0]$ through $\texttt{mem}[s-1]$. Each word is a w-bit string, often interpreted as the binary representation of an integer between 0 and $2^w - 1$. Each processor has r registers, labelled $\texttt{reg}[0]$ through $\texttt{reg}[r-1]$; each register contains one word. The computer also has space for i instructions; each processor has an instruction pointer.

Algorithms. A parallel algorithm is a sequence of instructions. (The algorithm fails on a computer of size p, s, w, r, i if it has more than i instructions.) Here are the possible instructions:

- Clear j: Set $\texttt{reg}[j] \leftarrow 0$. (The algorithm fails if $j \geq r$.)
- Increment j, k: Set $\texttt{reg}[j] \leftarrow (\texttt{reg}[k] + 1) \bmod 2^w$.
- Add j, k, ℓ: Set $\texttt{reg}[j] \leftarrow (\texttt{reg}[k] + \texttt{reg}[\ell]) \bmod 2^w$.
- Subtract j, k, ℓ: Set $\texttt{reg}[j] \leftarrow (\texttt{reg}[k] - \texttt{reg}[\ell]) \bmod 2^w$.
- Shift left j, k, ℓ: Set $\texttt{reg}[j] \leftarrow (\texttt{reg}[k] 2^{\texttt{reg}[\ell]}) \bmod 2^w$.
- Shift right j, k, ℓ: Set $\texttt{reg}[j] \leftarrow \left\lfloor \texttt{reg}[k] / 2^{\texttt{reg}[\ell]} \right\rfloor$.
- Word size j: Set $\texttt{reg}[j] \leftarrow w$.
- Identify j: Set $\texttt{reg}[j] \leftarrow$ this processor's number.
- Read j, k: Set $\texttt{reg}[j] \leftarrow \texttt{mem}[\texttt{reg}[k]]$. (The algorithm fails if $\texttt{reg}[k] \geq s$.)
- Write j, k: Set $\texttt{mem}[\texttt{reg}[k]] \leftarrow \texttt{reg}[j]$.
- Jump j, k, ℓ: If $\texttt{reg}[k] \geq \texttt{reg}[\ell]$, go to instruction j, rather than proceeding to the next instruction as usual.

A n-bit input to the algorithm is placed into memory at time 0, with n in the first word of memory, and the n bits packed into the next $\lceil n/w \rceil$ words of memory. (The algorithm fails if $2^w \leq n$, or if $s < 1 + \lceil n/w \rceil$.) All other words of memory, registers, etc. are set to 0. Each processor performs one instruction at time 1; each processor performs one instruction at time 2; and so on until all the processors have halted. The output is then in memory, encoded in the same way as the input.

Memory conflicts. During one time step, a single memory location might be accessed by more than one processor.

The **concurrent-read exclusive-write PRAM**, or **CREW PRAM**, allows any number of processors to simultaneously read the same memory location, but it does not allow two processors to simultaneously write to a single memory location.

The **concurrent-read common-concurrent-write PRAM**, or **Common CRCW PRAM**, allows any number of processors to simultaneously read the same memory location, and allows any number of processors to simultaneously write the same memory location, if they all write the same value. (If two processors attempt to write different values, the algorithm fails.)

A memory location cannot be read and written simultaneously.

Asymptotics. When we say that a parallel algorithm handles an n-bit input using (for example) time $O(n)$ on $O(n^3)$ processors with a word size of $O(\lg n)$ using $O(n^4)$ memory locations, we mean that it works on any sufficiently large computer: there are functions $p(n) \in O(n^3)$, $s(n) \in O(n^4)$, $w(n) \in O(\lg n)$, and $t(n) \in O(n)$ such that, for every n, every $p \geq p(n)$, every $s \geq s(n)$, every $w \geq w(n)$, every r larger than the highest register number used in the algorithm, every i larger than the length of the algorithm, and every input string of length n, the algorithm runs without failure on that input in time at most $t(n)$ on a parallel computer of size p, s, w, r, i.

When we do not mention the word size (and number of memory locations), we always mean that the required word size is logarithmic in the time-processor product (and, consequently, the required number of memory locations is polynomial in the time-processor product).

7. Modular exponentiation on the CREW PRAM

In this section we present a linear-time algorithm, using a polynomial number of processors, for modular exponentiation on the CREW PRAM.

Time $O(n(\lg n)^2)$. Given the binary representations of n-bit integers x and y, one can compute xy in time $O(\lg n)$ using $n^{O(1)}$ processors. One can compute x/y and $x \bmod y$ by Newton's method in time $O((\lg n)^2)$ using $n^{O(1)}$ processors. See [33, Theorem 12.1].

The base-2 exponentiation algorithm shown in Section 5, using these subroutines, computes $x^e \bmod m$ in time $O(n(\lg n)^2)$ if x, e, m have n bits. There are $O(n)$ iterations in the algorithm, each iteration involving a constant number of multiplications and divisions of $O(n)$-bit integers. The number of processors is polynomial in n.

Time $O(n \lg n)$. A division algorithm of Beame, Cook, and Hoover takes time only $O(\lg n)$ using $n^{O(1)}$ processors, after a precomputation taking time $O((\lg n)^2)$. See [3].

The main subroutine in the Beame-Cook-Hoover algorithm computes powers x^v, where x has n bits and $v \in \{0, 1, \ldots, n\}$, in time $O(\lg n)$. The idea is to use the CRT to recover x^v from the remainders $x^v \bmod p$ for enough small primes p. Beame, Cook, and Hoover precompute the primes p and the powers $u^v \bmod p$ for all $u \in \{0, 1, \ldots, p-1\}$ and all $v \in \{0, 1, \ldots, n\}$; then they can compute $x^v \bmod p$ as $(x \bmod p)^v \bmod p$. This is one of the ideas that we use in Section 8.

Time $O(n)$. To save another $\lg n$ factor, we choose an integer $b \geq 2$ so that $\lg b \in \Theta(\lg n)$, and we use the base-b exponentiation algorithm.

The number of iterations drops to $O(\log_b e) = O(n/\lg n)$. Each iteration involves computing $y^b x^v \bmod m$ for some $v \in \{0, 1, \ldots, b-1\}$; we compute y^b and x^v in time $O(\lg n)$ by the Beame-Cook-Hoover subroutine, then multiply, then use the Beame-Cook-Hoover algorithm again to divide by m.

8. Modular exponentiation on the Common CRCW PRAM

Fix $\epsilon > 0$. Some of the O constants below depend on ϵ.

In this section we present a sublinear-time algorithm, using a polynomial number of processors, for modular exponentiation on the Common CRCW PRAM:

Theorem 8.1. *There is an algorithm for the Common CRCW PRAM that, given the binary representations of positive integers x, e, and m, of total bit length n, computes the binary representation of $x^e \bmod m$ in time $O(n/\lg\lg n)$ using $O(n^{2+\epsilon})$ processors.*

The algorithm can be converted into a polynomial-size unbounded-fan-in circuit of depth $O(n/\lg\lg n)$, using the techniques explained in [32]. The algorithm of Section 7 can be converted into a polynomial-size bounded-fan-in circuit of depth $O(n)$ by the same techniques.

The only previous sublinear-time algorithms were algorithms that use many more processors or that drastically restrict m. See [14, Section 6] for a survey.

We do not know an exponentiation algorithm that takes time $O(n/\lg\lg n)$ using $n^{2+o(1)}$ processors. Allowing ϵ to vary with n would hurt the time bound: for example, choosing ϵ as $1/\lg\lg\lg\lg n$ would produce an algorithm that takes time $O(n\lg\lg\lg\lg n/\lg\lg n)$ using $O(n^{2+1/\lg\lg\lg\lg n})$ processors.

In a few applications, the exponent e is only a small part of the input, perhaps $\sqrt{n}$ or $\sqrt[3]{n}$ bits. The running time depends primarily on the length of e: the algorithm takes time $O((\lg e)/\lg\lg n) + (\lg n)^{O(1)}$ using $O(n^{2+\epsilon})$ processors.

How we save a $\lg\lg n$ factor. The algorithm in this section, like the simplified linear-time algorithm of Section 7, performs several multiplications before each reduction modulo m. It achieves better parallelization than the algorithm of Section 7 by

- avoiding the binary representation in the main loop—we use the explicit Chinese remainder theorem modulo m modulo p_j, as described in Section 4, to work consistently in the residue representation—and
- using the Cole-Vishkin parallel-addition algorithm to add $O(n)$ integers, each having $O(\lg n)$ bits, in time $O((\lg n)/\lg\lg n)$ using $n^{1+o(1)}$ processors. See [10] and [34].

Perhaps the same $\lg\lg n$ speedup can be obtained with other redundant representations of integers modulo m; we leave this exploration to the reader.

Outside the inner loop (step 6 below), the algorithm takes time only $(\lg n)^{O(1)}$ using $O(n^{2+\epsilon})$ processors. We take advantage of several standard FFT-based tools here:

- Given n bits representing two integers x, y (in binary), one can use the Schönhage-Strassen multiplication algorithm to compute the product xy in time $(\lg n)^{O(1)}$ using $n^{1+o(1)}$ processors. See [29].
- Given n bits representing two integers x, y with $y \neq 0$, one can compute the quotient $\lfloor x/y \rfloor$ in time $(\lg n)^{O(1)}$ using $n^{1+o(1)}$ processors. See [33, Theorem 12.1].
- Given n bits representing s integers, one can compute the product of all the integers in time $(\lg n)^{O(1)}$ using $n^{1+o(1)}$ processors, by multiplying pairs in parallel.
- Given n bits representing integers $u, p_1, p_2, \ldots, p_s$, one can use the Borodin-Moenck remainder-tree algorithm to compute $u \bmod p_1, \ldots, u \bmod p_s$ in time $(\lg n)^{O(1)}$ using $n^{1+o(1)}$ processors. See [22], [8, Sections 4–6], and [6, Section 18].

Without the FFT-based tools, step 4 below requires roughly n^3 processors to be carried out in polylogarithmic time. This is (after some serialization) adequate for Theorem 8.1, but it becomes a bottleneck when e is much shorter than m.

Step 1: build multiplication tables. Fix a positive rational number $\delta \leq \epsilon/3$. Compute an integer $a \geq 1$ within 1 of $\delta \lg n$. Note that an a-bit integer fits into $O(1)$ words, since $a \in O(\lg n)$.

Compute xy and (for $y \neq 0$) $\lfloor x/y \rfloor$ for each pair x, y of a-bit integers in parallel. Store the results in a table of $O(2^{2a})$ words. This takes time $O(a)$ using 2^{2a} processors; i.e., time $O(\lg n)$ using $O(n^{2\delta})$ processors.

Note that, if x and y are $O(\lg n)$-bit integers, then x and y are also $O(a)$-bit integers, so one processor can compute $x+y$, $x-y$, xy, and $\lfloor x/y \rfloor$ in time $O(1)$ with the help of this table.

Step 2: find primes. Define $b = 2^{3a}$. Note that $b \in \Theta(n^\delta)$.

Find the smallest integer $k \geq 6$ such that $2^k \geq 4 + 4b(n+k)$. Use a parallel sieve, as described in [30], to find the primes $p_1, p_2, \ldots, p_s$ between 2 and 2^k. This takes time $O((1+\delta)\lg n)$ using $O(n^{1+\delta})$ processors; note that $s \leq p_s \in O(n^{1+\delta})$.

Define $P = p_1 p_2 \cdots p_s$. Note that $\lg P \in O(n^{1+\delta})$ by, e.g., [28, Theorem 9]. Note also that $P \geq 4(m^2 \sum p_i)^b$. (Indeed, $2^k \geq 41$, so $\log P = \log p_1 + \log p_2 + \cdots + \log p_s \geq 2^k(1 - 1/\log 2^k)$ by [28, Theorem 4]; also $p_1 + p_2 + \cdots + p_s \leq 2^{2k}$. Thus $\lg P \geq \log P \geq 2^{k-1} \geq 2 + b(2n+2k) \geq \lg 4 + b(\lg m^2 + \lg \sum p_i)$.)

Multiply $p_1, \ldots, p_s$ to compute P. This takes time $(\lg P)^{O(1)}$ using $(\lg P)^{1+o(1)}$ processors; i.e., time $(\lg n)^{O(1)}$ using $n^{1+\delta+o(1)}$ processors.

Step 3: build power tables. For each $i \in \{1, 2, \ldots, s\}$ in parallel, for each $u \in \{0, 1, \ldots, p_i - 1\}$ in parallel, for each $v \in \{0, 1, \ldots, b\}$ in parallel (or serial), compute $u^v \bmod p_i$. Store the results in a table. This takes time $O(\lg b)$ using $O(sp_s(b+1))$ processors; i.e., time $O(\lg n)$ using $O(n^{2+2\delta} \lg n)$ processors.

(An alternative approach, with smaller tables, is to find a generator g_i for the multiplicative group $(\mathbf{Z}/p_i)^*$, build a table of powers of g_i, and build a table of discrete logarithms base g_i. See [30].)

Step 4: compute ECRT coefficients. For each i in parallel, compute $P_i = P/p_i$; $P_i \bmod p_i$; $P_i \bmod m$; and $P_i \bmod m \bmod p_1, P_i \bmod m \bmod p_2, \ldots, P_i \bmod m \bmod p_s$. This takes time $(\lg P)^{O(1)}$ using $s(\lg P)^{1+o(1)}$ processors; i.e., time $(\lg n)^{O(1)}$ using $n^{2+2\delta+o(1)}$ processors.

Next, for each i in parallel, compute $q_i = (P_i \bmod p_i)^{p_i - 2} \bmod p_i$, so that q_i is the inverse of P_i modulo p_i. This takes time $O(\lg p_s)$ using s processors; i.e., time $O(\lg n)$ using $O(n^{1+\delta})$ processors.

Step 5: convert to the residue representation. Set $x \leftarrow x \bmod m$. This takes time $(\lg n)^{O(1)}$ using $n^{1+o(1)}$ processors.

Set $x_1 \leftarrow x \bmod p_1, x_2 \leftarrow x \bmod p_2, \ldots, x_s \leftarrow x \bmod p_s$. This takes time $(\lg n)^{O(1)}$ using $n^{1+\delta+o(1)}$ processors. Also set $y_1 \leftarrow 1, y_2 \leftarrow 1, \ldots, y_s \leftarrow 1$.

Let l be the number of base-b digits in e. Write $e = \sum_{k=0}^{l-1} e_k b^k$ with $0 \leq e_k < b$. Note that $l \in O(n/\lg n)$.

Step 6: inner loop. Repeat the following substeps for $k \leftarrow l-1$, $k \leftarrow l-2$, $\ldots$, $k \leftarrow 0$. Each substep will take time $O((\lg n)/\lg\lg n)$, so the total time here is $O(n/\lg\lg n)$.

Invariant: The integer y represented by $(y_1, y_2, \ldots, y_s)$ is between $-m\sum_i p_i$ and $m\sum_i p_i$, and is congruent to $x^{e_{l-1}b^{l-k-2} + \cdots + e_{k+2}b + e_{k+1}}$ modulo m. The plan is to compute the residue representation of $u = y^b x^{e_k}$, and then change y to the integer v identified in Theorem 3.1. Note that $|u| < (m^2 \sum_i p_i)^b \leq P/4$; hence $v \equiv u \equiv x^{e_{l-1}b^{l-k-1} + \cdots + e_{k+2}b^2 + e_{k+1}b + e_k} \pmod{m}$.

For each i in parallel, compute $t_i \leftarrow (y_i^b \bmod p_i)(x_i^{e_k} \bmod p_i)q_i \bmod p_i$. This takes time $O(1)$ using s processors, thanks to the precomputed power table.

Compute $r \leftarrow \operatorname{round} \sum_i t_i/p_i$ with the help of Theorem 2.2. The fixed-point divisions take time $O(1)$ using s processors. The addition takes time $O((\lg s)/\lg\lg s)$ using $s^{1+o(1)}$ processors.

For each j in parallel, compute $y_j \leftarrow \sum_i t_i(P_i \bmod m \bmod p_j)$. This takes time $O((\lg s)/\lg\lg s)$ using $s^{2+o(1)}$ processors; i.e., time $O((\lg n)/\lg\lg n)$ using $n^{2+2\delta+o(1)}$ processors.

For each j in parallel, compute $y_j \leftarrow (y_j - r(P \bmod m \bmod p_j)) \bmod p_j$. This takes time $O(1)$ using s processors.

Step 7: convert to the binary representation. Use Theorem 2.1 to compute the binary representation of the integer y whose residue representation is $(y_1, y_2, \ldots, y_s)$. This takes time $(\lg n)^{O(1)}$ using $n^{2+2\delta+o(1)}$ processors.

Now $x^e = x^{e_{l-1}b^{l-1}+\cdots+e_1 b+e_0} \equiv y \pmod m$. The output of the algorithm is y.

References

[1] A. Agashe, K. Lauter, R. Venkatesan, *Constructing elliptic curves with a known number of points over a prime field*. Available from `http://research.microsoft.com/~klauter/`.

[2] Manindra Agrawal, Neeraj Kayal, Nitin Saxena, *PRIMES is in P* (2002). Available from `http://www.cse.iitk.ac.in/news/primality.html`.

[3] Paul W. Beame, Stephen A. Cook, H. James Hoover, *Log depth circuits for division and related problems*, SIAM Journal on Computing **15** (1986), 994–1003. ISSN 0097–5397.

[4] Daniel J. Bernstein, *Detecting perfect powers in essentially linear time, and other studies in computational number theory*, Ph.D. thesis, University of California at Berkeley, 1995.

[5] Daniel J. Bernstein, *Multidigit modular multiplication with the explicit Chinese remainder theorem*, in [4] (1995). Available from `http://cr.yp.to/papers.html`.

[6] Daniel J. Bernstein, *Fast multiplication and its applications*. Available from `http://cr.yp.to/papers.html`.

[7] Daniel J. Bernstein, *Pippenger's exponentiation algorithm*, draft. Available from `http://cr.yp.to/papers.html`.

[8] Allan Borodin, Robert T. Moenck, *Fast modular transforms*, Journal of Computer and System Sciences **8** (1974), 366–386; older version, not a subset, in [22]. ISSN 0022–0000. MR 51:7365. Available from `http://cr.yp.to/bib/entries.html#1974/borodin`.

[9] Henri Cohen, *A course in computational algebraic number theory*, Graduate Texts in Mathematics, 138, Springer-Verlag, Berlin, 1993. ISBN 3–5440–55640–0. MR 94i:11105.

[10] Richard Cole, Uzi Vishkin, *Faster optimal parallel prefix sums and list ranking*, Information and Computation **81** (1989), 334–352. ISSN 0890–5401. MR 90k:68056.

[11] Jean-Marc Couveignes, *Computing a square root for the number field sieve*, in [20] (1993), 95–102.

[12] Shauna M. Meyer Eikenberry, Jonathan P. Sorenson, *Efficient algorithms for computing the Jacobi symbol*, Journal of Symbolic Computation **26** (1998), 509–523. ISSN 0747–7171.

[13] Faith E. Fich, *The complexity of computation on the parallel random access machine*, in [27] (1993), 843–899.

[14] Daniel M. Gordon, *A survey of fast exponentiation methods*, Journal of Algorithms **27** (1998), 129–146. ISSN 0196–6774. MR 99g:94014. Available from `http://www.ccrwest.org/gordon/dan.html`.

[15] Richard M. Karp (chairman), *13th annual symposium on switching and automata theory*, IEEE Computer Society, Northridge, 1972.

[16] Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 1st edition, 1st printing, Addison-Wesley, Reading, 1969; see also newer version in [17]. MR 44:3531.

[17] Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 1st edition, 2nd printing, Addison-Wesley, Reading, 1971; see also older version in [16]; see also newer version in [18]. MR 44:3531.

[18] Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 2nd edition, Addison-Wesley, Reading, 1981; see also older version in [17]; see also newer version in [19]. ISBN 0–201–03822–6. MR 83i:68003.

[19] Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 3rd edition, Addison-Wesley, Reading, 1997; see also older version in [18]. ISBN 0–201–89684–2.

[20] Arjen K. Lenstra, Hendrik W. Lenstra, Jr. (editors), *The development of the number field sieve*, Lecture Notes in Mathematics, 1554, Springer-Verlag, Berlin, 1993. ISBN 3–540–57013–6. MR 96m:11116.

[21] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone, *Handbook of applied cryptography*, CRC Press, Boca Raton, Florida, 1996. ISBN 0–8493–8523–7. MR 99g:94015. Available from `http://cacr.math.uwaterloo.ca/hac`.

[22] Robert T. Moenck, Allan Borodin, *Fast modular transforms via division*, in [15] (1972), 90–96; newer version, not a superset, in [8]. Available from `http://cr.yp.to/bib/entries.html#1972/moenck`.

[23] Peter L. Montgomery, *An FFT extension of the ellpitic curve method of factorization*, Ph.D. thesis, University of California at Los Angeles, 1992.

[24] Peter L. Montgomery, Robert D. Silverman, *An FFT extension to the $P-1$ factoring algorithm*, Mathematics of Computation **54** (1990), 839–854. ISSN 0025–5718. MR 90j:11142.

[25] Andrew M. Odlyzko, Gary Walsh, Hugh Williams (editors), *Conference on the mathematics of public key cryptography: the Fields Institute for Research in the Mathematical Sciences, Toronto, Ontario, June 12–17, 1999*, book of preprints distributed at the conference, 1999.

[26] Christos M. Papadimitriou, *Computational complexity*, Addison-Wesley, Reading, Massachusetts, 1994. ISBN 0201530821. MR 95f:68082.

[27] John H. Reif (editor), *Synthesis of parallel algorithms*, Morgan Kaufman, San Mateo, California, 1993. ISBN 1–55860–135–X. MR 94c:68086.

[28] J. Barkley Rosser, Lowell Schoenfeld, *Approximate formulas for some functions of prime numbers*, Illinois Journal of Mathematics **6** (1962), 64–94. ISSN 0019–2082. MR 25:1139. Available from `http://cr.yp.to/bib/entries.html#1962/rosser`.

[29] Arnold Schönhage, Volker Strassen, *Schnelle Multiplikation großer Zahlen*, Computing **7** (1971), 281–292. ISSN 0010–485X. MR 45:1431. Available from `http://cr.yp.to/bib/entries.html#1971/schoenhage-mult`.

[30] Jonathan P. Sorenson, Ian Parberry, *Two fast parallel prime number sieves*, Information and Computation **144** (1994), 115–130. ISSN 0890–5401. MR 95h:11097.

[31] Jonathan P. Sorenson, *A sublinear-time parallel algorithm for integer modular exponentiation*, in [25] (1999).

[32] Larry Stockmeyer, Uzi Vishkin, *Simulation of parallel random access machines by circuits*, SIAM Journal on Computing **13** (1984), 409–422. ISSN 0097–5397. MR 85g:68018.

[33] Stephen R. Tate, *Newton iteration and integer division*, in [27] (1993), 539–572.

[34] Uzi Vishkin, *Advanced parallel prefix-sums, list ranking and connectivity*, in [27] (1993), 215–257.

Department of Mathematics, Statistics, and Computer Science (M/C 249), The University of Illinois at Chicago, Chicago, IL 60607–7045

E-mail address: `djb@cr.yp.to`

Department of Computer Science and Software Engineering, Butler University, 4600 Sunset Avenue, Indianapolis, IN 46208

E-mail address: `sorenson@butler.edu`