Batch Integer Smooth-Part Find

November 28, 2016

Batch Integer Smooth-Part Find (will be referred to from now as Batch-Find or BSF) is a variation of Daniel J. Bernstein's Batch Integer Division (also designed by DJB) that, given a finite sequence of integers, S, and a finite set of primes P, finds the of each integer in S with respect to the set P [5].

The algorithm finds the "smooth-part" of integers in our set, but what is a "smooth-part?"The smooth part of an integer with respect to a set of primes is the largest divisor of that integer that divides over the set of primes (which means that divisor is P-smooth). For instance:

Let's suppose a number's prime factorization is $2^3 5^1 7^1 23^1$.
Given the set of primes P $= \{2, 3, 5\}$, the P-smooth-part of the above number is $2^3 5^1$.

In the context of Index Calculus, BSF provides an efficient method of data collection in phase 1 of Index Calculus. When used with regular Index Calculus we can test if candidates of g^x are factorable over our factor base in large batches. Additionally we can use BSF for testing r (and s values if a smooth candidate is found from the batch of r values) from the Modified Extended Euclidean Algorithm for smoothness for further speed up in phase 1.

We ended up not using BSF due to timing discrepancy between its implementation and that of the Linear Sieve method (additionally BSF can't be pipelined with the Linear Sieve method like MEEA). BSF was tested in combination with MEEA successfully with smaller moduli on a single thread machine, but was not tested on our main platform.

0.1 The subroutines

Because BSF does its computations in batches, several subroutines are used to take advantage of the batch structure.

1. Product Tree

   - Due to CPU and compiler optimization, on most CPUs multiplication is a constant time process as long as the multiplier and multiplicand can each fit in a single register. However once numbers become too large and some bignum library is required, multiplication is then almost completely "handled by software", and multiplication actually becomes an expensive operation that runs in $n \times m$ time where n and m are the number of digits in the multiplier and multiplicand. Time spent on multiplication is referred to as $\mu - time$.

- When dealing with the product of a set of integers a product tree is used to avoid the having to sequentially multiply over and over again throughout the entire set.
- Conceptually, the product tree is very simple. The product tree is a binary tree where the leaves of the product tree are the numbers in the set we wish to multiply. Each non-leaf node of the tree holds the product of its two children, with the root of the tree being the product of every number in the set.
- Computation of the product tree of a set of elements: $\{p_1, p_2 \ ...p_m\}$ takes at most $kb\mu$ time, where k is the smallest integer such that $2^k \geq m$, b = number of bits in $\{p_1 \ ... \ p_m\}$, and $\mu = \mu - time$ [6].

2. Remainder Tree

- Similar to the the product tree, computing an integer $h \pmod{x_k}$ for $k \in \{1...n\}$ can be a costly process when done sequentially done over the set of $\{x_1...x_n\}$, and can be sped up using a binary tree structure.
- A remainder tree is binary tree where the leaves are h modulo each element of $\{x_1...x_n\}$, and every nonleaf node is h modulo the product of its two children.
- Unlike a product tree, where we construct the tree bottom-up, we construct the remainder tree top-down, using the product tree of $\{x_1...x_n\}$ to get the intermediate products to use as the moduli in the non-leaf nodes of the remainder tree.
- Computation of the remainder tree of a set of elements, using the product tree of $\{x_1...x_n\}$ to get the intermediate products to use as the moduli in the non-leaf nodes of the remainder tree.
- Computation of the remainder tree as described above takes $O(b(lgb)^2 lglgb)$ time where b = number of bits in $\{x_1...x_n\}$ [5][8].

0.2 The algorithm

Let:
$P = \{p_1, p_2, p_3 ... p_m\}$ be a set of distinct prime numbers
$S = \{x_1, x_2, x_3 ... x_n\}$ be a set of positive integers

1. Compute $z = p_1 \times p_2 \times p_3 \times \ ... \ \times p_m$ using a Product Tree
2. Compute $z \pmod{x_1}$ ... $z \pmod{x_n}$ using a Remainder Tree
3. For each $k \in \{1 \ ... \ n\}$: Compute $y_k \leftarrow (z \pmod{x_k})^{2^e} \pmod{x_k}$ by repeated squaring, where e is the smallest integer such that $2^{2^e} \geq x_k$
4. For each $k \in \{1 \ ... \ n\}$: Print $\{x_k, y_k\}$. These are the smooth parts of each x_k

0.3 Possible improvements for BSF

The basic method of BSF as it turns out provides more information than we want for our problem. BSF generates the smooth parts of integers (The "portion" of the integer that fulfills the smoothness requirement defined by our factor base), but we only care if the entire number fulfills our smoothness requirement (finding the smooth part of integers has applications outside of Index Calculus). Thus we can skip step 4 of the algorithm entirely and simply compare each y_k generated from step 3 to the corresponding x_k in our batch of candidates. If they're equal, then x_k divides by our factor base. If we had continued with step 4, then $gcd(x_k, y_k)$ would have been x_k, meaning the smooth part of x_k would've been itself.

0.4 Analysis of BSF

Let b be the number of bits in the total input (the collective number of bits in the batch of candidates). The runtime for each step is as follows:

1. Step 1 takes $O(b(lgb)^2 lglgb)$ [8]
2. Step 2 takes $O(b(lgb)^2 lglgb)$ [8]
3. Step 3 takes $O(b_k(lgb)^2 lglgb)$ where b_k is the number of bits in x_k, since e is bounded by lgb. Step 3 run over the entire batch would therefore take $O(b(lgb)^2 lglgb)$
4. Step 4 we were able to avoid doing altogether due to the goal of our algorithm.

Altogether the algorithm takes

$$O(b(lgb)^2 lglgb)$$

time.