When File Synchronization Meets Number Theory

Antoine Amarilli, Fabrice Ben Hamouda, Florian Bourse,
Robin Morisset, David Naccache, and Pablo Rauzy

École normale supérieure, Département d'informatique
45, rue d'Ulm, F-75230, Paris Cedex 05, France.
surname.name@ens.fr (except for fabrice.ben.hamouda@ens.fr)

Abstract. This work revisits *set reconciliation*, a problem consisting in synchronizing two multisets while minimizing the amount of data transmitted. We propose a new number theoretic reconciliation protocol called "Divide & Factor". Like prior proposals, our protocol in optimal in terms of asymptotic transmission complexity. Nonetheless, this new protocol offers interesting parameter trade-offs resulting in experimentally measured *constant-factor* transmission gains over the popular software `rsync`. Reconciliation experiments show that the new protocol usually transmits less data than `rsync` at the expense of lengthier calculations.

1 Introduction

This work revisits *set reconciliation*, a problem consisting in synchronizing two multisets while minimizing the amount of data transmitted. Set reconciliation arises in many practical situations, the most typical of which is certainly incremental backups performed over a slow network link.

Set reconciliation has already several efficient and elegant solutions. For instance, [3] presents a particularly interesting reconciliation protocol whose computational and communication complexities are linear in the number of differences between the reconciled multisets, which mathches the obvious lower bound.

We refer the reader to [3,4,5] (to quote a few references) for more on the problem's history and its existing solutions.

This article proposes a new reconciliation protocol based on number theory. In terms of asymptotic transmission complexity, the proposed procedure reaches optimality, like prior proposals. Nonetheless, the new protocols offer interesting parameter trade-offs resulting in experimentally measured *constant-factor* gains over the popular software `rsync`.

We observed during most of our reconciliation experiments that the new protocol transmitted less bytes than `rsync` but required lengthier calculations.

Beyond these practical concerns the mathematical ideas underlying this work seem new and interesting as such.

TODO: announce the structure of the rest of the paper.

2 "Divide & Factor" Set Reconciliation

2.1 Problem Definition and Notations

Oscar possesses an old version of a directory $\mathfrak{D}$ that he wishes to update. Neil has the new, up-to-date version $\mathfrak{D}'$: $\mathfrak{D}$ and $\mathfrak{D}'$ can differ both in their files and in their tree structure. Oscar

wishes to obtain knowledge about $\mathfrak{D}'$ but wishes to *exchange as little data as possible* during the synchronization process.

To tackle this problem we separate the *"what"* from the *"where"* and disregard the relative position of files. We represent $\mathfrak{D}$ as a multiset of files which we denote as $\mathfrak{F} = \{F_0, \ldots, F_n\}$, and likewise represent $\mathfrak{D}'$ as $\mathfrak{F}' = \{F'_0, \ldots, F'_{n'}\}$.

Let t_0 be the number of discrepancies between $\mathfrak{F}$ and $\mathfrak{F}'$ that Oscar wishes to learn, *i.e.*:

$$t_0 = \#\mathfrak{F} + \#\mathfrak{F}' - 2\#\left(\mathfrak{F} \bigcap \mathfrak{F}'\right) = \#\left(\mathfrak{F} \bigcup \mathfrak{F}'\right) - \#\left(\mathfrak{F} \bigcap \mathfrak{F}'\right)$$

Given a file F, we denote by $\texttt{Hash}(F)$ its image by a collision-resistant hash function such as SHA-1. Let $\texttt{HashPrime}(F)$[1] be a function hashing files (uniformly) into primes smaller than 2^u. [TODO(antoine): what is the definition of u, is it something like "there exists u such that HashPrime hashes to primes smaller than 2^u"?]. Define the shorthand notations: $h_i = \texttt{HashPrime}(F_i)$ and $h'_i = \texttt{HashPrime}(F'_i)$.

2.2 Description of the Basic Exchanges

The quantity t_0 is unknown to Oscar and Neil. However, in our protocol, Oscar and Neil will assume that t_0 is smaller than some t and attempt to perform synchronization. If they are right, synchronization will succeed; if they are wrong, they will transmit more information and try again, as will be explained later.

We generate a prime p such that:

$$2^{2ut+1} \leq p < 2^{2ut+2} \tag{1}$$

Given $\mathfrak{F}$, Oscar generates and sends to Neil the redundancy:

$$c = \prod_{F_i \in \mathfrak{F}} \texttt{HashPrime}(F_i) = \prod_{i=1}^{n} h_i \bmod p$$

Neil computes:

$$c' = \prod_{F'_i \in \mathfrak{F}'} \texttt{HashPrime}(F'_i) = \prod_{i=1}^{n'} h'_i \bmod p \quad \text{and} \quad s = \frac{c'}{c} \bmod p$$

Using [7] the integer s can be written as:

$$s = \frac{a}{b} \bmod p \text{ where the } G_i \text{ denote files and } \begin{cases} a = \prod\limits_{G_i \in \mathfrak{F}' \wedge G_i \notin \mathfrak{F}} \texttt{HashPrime}(G_i) \\ b = \prod\limits_{G_i \notin \mathfrak{F}' \wedge G_i \in \mathfrak{F}} \texttt{HashPrime}(G_i) \end{cases}$$

Note that if our assumption $t_0 \leq t$ is correct $\mathfrak{F}$ and $\mathfrak{F}'$ differ by at most t elements and a and b are strictly less than 2^{ut}. Theorem 1 (see [2]) guarantees a and b can be efficiently recovered from s: this problem is known as the *Rational Number Reconstruction* [4,8] and is typically solved using Gauss' algorithm for finding the shortest vector in a bi-dimensional lattice [7].

[1] We discuss the design of `HashPrime` in Appendix C.

Theorem 1. *Let $a, b \in \mathbb{Z}$ such that $-A \leq a \leq A$ and $0 < b \leq B$. Let $p > 2AB$ be a prime and $s = ab^{-1} \mod p$. Then a, b can be recovered from A, B, s, p in polynomial time.*

Taking $A = B = 2^{ut} - 1$, (1) implies that $2AB < p$. Moreover, $0 \leq a \leq A$ and $0 < b \leq B$. Thus Oscar can recover a and b from s in polynomial time. By testing the divisibility of a and b by the h_i and the h'_i, Neil and Oscar can deterministically identify the discrepancies between $\mathfrak{F}$ and $\mathfrak{F}'$ and settle them.

Formally, this is done as follows:

Oscar		**Neil**
computes c		
	$\xrightarrow{c}$	
		computes a, b
		if a doesn't factor into a product of h'_is
		then output $\perp$ and halt
		else let $\mathfrak{S} = \{F'_i \text{ s.t. } a \bmod h'_i = 0\}$
	$\xleftarrow{\mathfrak{S},b}$	
deletes files s.t. $b \bmod h_i = 0$		
adds $\mathfrak{S}$ to the disk		

As we have just seen, the "output $\perp$ and halt" protocol interruption should never occur if indeed $t_0 \leq t$. However, a file synchronization procedure that works *only* for a limited number of differences is not really useful in practice. Section 2.3 explains how to extend the protocol even when the number of differences exceeds t, the informational capacity of the modulus p.[TODO(antoine): I'm not I understand why if t is too small we cannot have that a factors into a product of h'_is even though the protocol has actually failed and b is not what we expect. Will this really *never* occur, or is it very unlikely?]

2.3 The Case of Insufficient Information

To extend the protocol to an arbitrary t_0, Oscar and Neil agree on an infinite set of primes $p_1, p_2, \ldots$ As long as the protocol fails, Neil will keep accumulating information about the difference between $\mathfrak{F}$ and $\mathfrak{F}'$ as shown in Appendix A. Note that no information is lost and that the transmitted modular knowledge about the difference adds up until it reaches a threshold sufficient to reconcile $\mathfrak{F}$ and $\mathfrak{F}'$.

3 Transmission Complexity

This section explores two strategies to reduce the size of p and hence improve transmission by *constant factors* (from an asymptotic communication standpoint, improvements cannot be expected as the protocol already transmits information proportional to t, the difference to settle). [TODO(antoine): This claim is not entirely obvious to me. I think it would be worthwhile to do some simple accounting to give the size of everything that is transferred so that everyone can see that we are indeed linear in the size of the symmetric difference.]

3.1 Probabilistic Decoding: Reducing p

Generate a prime p about twice shorter than the p recommended in section 2.2, namely:

$$2^{ut+w-1} < p \leq 2^{ut+w} \tag{2}$$

where $w \geq 1$ is some small integer (say $w = 50$). Let $\eta = \max(n, n')$. The new redundancy c is calculated as previously and is hence also approximately twice smaller. Namely:

$$s = \frac{a}{b} \bmod p \text{ and } \begin{cases} a = \prod_{G_i \in \mathfrak{F}' \wedge G_i \notin \mathfrak{F}} \texttt{HashPrime}(G_i) \\ b = \prod_{G_i \notin \mathfrak{F}' \wedge G_i \in \mathfrak{F}} \texttt{HashPrime}(G_i) \end{cases}$$

and since there are at most t differences, we must have:

$$ab \leq 2^{ut} \tag{3}$$

By opposition to section 2.2 we do not have a fixed bound for a and b anymore; equation (3) only provides a bound for the *product* ab. Therefore, we define a sequence of at most $\lceil ut/w \rceil + 1$ couples of bounds:

$$(A_i, B_i) = \left(2^{wi}, \left\lfloor \frac{p-1}{2^{wi+1}} \right\rfloor\right) \quad \text{where} \quad B_i > 1 \quad \text{and} \quad \forall i > 0,\ 2A_iB_i < p$$

Equations (2) and (3) imply that there must exist at least one index i such that $0 \leq a \leq A_i$ and $0 < b \leq B_i$. Then using Theorem 1, given (A_i, B_i, p, s) one can recover (a, b), and hence the difference between $\mathfrak{F}$ and $\mathfrak{F}'$.

The problem is that (unlike section 2.2) we have no guarantee that such an (a, b) is unique. Namely, we could (in theory) stumble over an $(a', b') \neq (a, b)$ satisfying (3) for some index $i' \neq i$. We expect this to happen with negligible probability (that we do not try to estimate here) when w is large enough, but this makes the modified protocol heuristic only. [TODO(antoine): Si je comprends bien, vous améliorez la complexité, mais en supposant que les choses se passent bien (et vous dites que c'est très probablement le cas). J'ai l'impression que c'est quand même un peu malhonnête de ne pas avoir un résultat sur l'*espérance* de la quantité d'information à transmettre, en prenant aussi en compte les cas où les choses se passent mal ; ou bien, si on ne peut pas avoir de tel résultat, il faudrait avouer clairement qu'on conjecture juste que c'est meilleur en moyenne. Non ?]

3.2 The File Laundry: Reducing u

What happens if we brutally shorten u in the basic Divide & Factor protocol?

As expected by the birthday paradox, we should start seeing collisions. Let us analyze the statistics governing the appearance of collisions.

Consider `HashPrime` as a random function from $\{0,1\}^*$ to $\{0, \ldots, 2^u - 1\}$. Let X_i be the random variable:

$$X_i = \begin{cases} 1 & \text{if file } F_i \text{ collides with another file.} \\ \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, we have $\Pr[\, X_i = 1 \,] \leq \frac{\eta-1}{2^u}$. The average number of colliding files is hence:

$$\mathbb{E}\left[\sum_{i=0}^{\eta-1} X_i\right] \leq \sum_{i=0}^{\eta-1} \frac{\eta-1}{2^u} = \frac{\eta(\eta-1)}{2^u}$$

For instance, for $\eta = 10^6$ files and 32-bit digests, the expected number of colliding files is less than 233.

However, it is important to note that a collision can only yield a *false positive*, and never a *false negative*. In other words, while a collision may obliviate a difference[2] a collision will never create a nonexistent difference *ex nihilo*.

Thus, it suffices to replace `HashPrime`(F) by a diversified `HashPrime`$(k|F)$ to quickly filter-out file differences by repeating the protocol for $k = 1, 2, \ldots$ At each iteration the parties will detect new files and new deletions, fix these and "launder" again the remaining multisets.

Assume that the diversified `HashPrime`$(k|F)$'s are random and independent. To understand why the probability that a stubborn file persists colliding decreases exponentially with the number of iterations k, assume that η remains invariant between iterations and define the following random variables:

$$X_i^\ell = \begin{cases} 1 & \text{if file } F_i \text{ collides with another file during iteration } \ell. \\ \\ 0 & \text{otherwise.} \end{cases}$$

$$Y_i = \prod_{\ell=1}^{k} X_i^\ell = \begin{cases} 1 & \text{if file } F_i \text{ collides with another file during the } k \text{ first protocol iterations.} \\ \\ 0 & \text{otherwise.} \end{cases}$$

By independence, we have:

$$\Pr[\, Y_i = 1 \,] = \prod_{\ell=1}^{k} \Pr\left[\, X_i^\ell = 1 \,\right] = \Pr[\, X_i^1 = 1 \,] \ldots \Pr\left[\, X_i^k = 1 \,\right] \leq \left(\frac{\eta-1}{2^u}\right)^k$$

Therefore the average number of colliding files is:

$$\mathbb{E}\left[\sum_{i=0}^{\eta-1} Y_i\right] \leq \sum_{i=0}^{\eta-1} \left(\frac{\eta-1}{2^u}\right)^k = \eta\left(\frac{\eta-1}{2^u}\right)^k$$

And the probability that at least one false positive will survive k rounds is:

[2] *e.g.* make the parties blind to the difference between `index.htm` and `iexplore.exe`.

$$\epsilon_k \leq \eta \left(\frac{\eta - 1}{2^u} \right)^k$$

For the previously considered instance[3] we get $\epsilon_2 \leq 5.43\%$ and $\epsilon_3 \leq 2 \cdot 10^{-3}\%$.

A more refined (but somewhat technical) analysis. As mentioned previously, the parties can remove the files confirmed as different during iteration k and work during iteration $k+1$ only with common and colliding files. Now, the only collisions that can fool round k, are the collisions of file-pairs (F_i, F_j) such that F_i and F_j have both already collided during *all the previous iterations*[4]. We call such collisions "masquerade balls". Define the two random variables:

$$Z_i^\ell = \begin{cases} 1 & \text{if } F_i \text{ participated in masquerade balls during all } \ell \text{ first protocol iterations.} \\ 0 & \text{otherwise.} \end{cases}$$

$$X_{i,j}^\ell = \begin{cases} 1 & \text{if files } F_i \text{ and } F_j \text{ collide during iteration } \ell. \\ 0 & \text{otherwise.} \end{cases}$$

Inclure ici image du bal masque

Set $Z_i^0 = 1$ and write $p_\ell = \Pr\left[Z_i^{\ell-1} = 1 \text{ and } Z_j^{\ell-1} = 1 \right]$ for all ℓ and $i \neq j$. For $k \geq 1$, we have:

$$\begin{aligned} \Pr\left[Z_i^k = 1 \right] &= \Pr\left[\exists j \neq i,\, X_{i,j}^k = 1,\, Z_i^{k-1} = 1 \text{ and } Z_j^{\ell-1} = 1 \right] \\ &\leq \sum_{j=0, j\neq i}^{\eta-1} \Pr\left[X_{i,j}^{k-1} = 1 \right] \Pr\left[Z_i^{k-1} = 1 \text{ and } Z_j^{k-1} = 1 \right] \\ &\leq \frac{\eta - 1}{2^u} p_{k-1} \end{aligned}$$

Furthermore $p_0 = 1$ and

$$\begin{aligned} p_\ell &= \Pr\left[X_0^\ell = X_1^\ell,\, Z_0^\ell = 1 \text{ and } Z_1^\ell = 1 \right] + \Pr\left[X_0^\ell \neq X_1^\ell,\, Z_0^\ell = 1 \text{ and } Z_1^\ell = 1 \right] \\ &\leq \Pr\left[X_0^\ell = X_1^\ell,\, Z_0^{\ell-1} = 1 \text{ and } Z_1^{\ell-1} = 1 \right] \\ &\quad + \sum_{i\geq 2, j\geq 2} \Pr\left[X_{0,i}^\ell = 1,\, X_{1,j}^\ell = 1,\, Z_0^{\ell-1} = 1 \text{ and } Z_1^{\ell-1} = 1 \right] \\ &= \Pr\left[X_0^\ell = X_1^\ell \right] \Pr\left[Z_0^{\ell-1} = 1 \text{ and } Z_1^{\ell-1} = 1 \right] \\ &\quad + \sum_{i\geq 2, j\geq 2} \Pr\left[X_{0,i}^\ell = 1 \right] \Pr\left[X_{1,j}^\ell = 1 \right] \Pr\left[Z_0^{\ell-1} = 1 \text{ and } Z_1^{\ell-1} = 1 \right] \\ &\leq \frac{1}{2^u} p_{\ell-1} + \frac{(\eta-2)^2}{2^{2u}} p_{\ell-1} = p_{\ell-1} \left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}} \right) \end{aligned}$$

[3] $\eta = 10^6$,$u = 32$.

[4] Note that we do not require that F_i and F_j repeatedly collide *which each other*. *e.g.* we may witness during the first round $\hbar_{1,u}(F_1) = \hbar_{1,u}(F_2)$ and $\hbar_{1,u}(F_3) = \hbar_{1,u}(F_4)$ while during the second round $\hbar_{2,u}(F_1) = \hbar_{2,u}(F_4)$ and $\hbar_{2,u}(F_2) = \hbar_{2,u}(F_3)$.

hence:

$$p_\ell \leq \left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}}\right)^\ell,$$

and

$$\Pr\left[\, Z_i^\ell = 1 \,\right] \leq \left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}}\right)^{k-1}$$

And finally, the survival probability of at least one false positive after k iterations satisfies:

$$\epsilon'_k \leq \frac{\eta(\eta-1)}{2^u}\left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}}\right)^{k-1}$$

For $(\eta = 10^6, u = 32, k = 2)$, we get $\epsilon'_2 \leq 0.013\%$.

How to select u? For a fixed k, ϵ'_k decreases as u grows. For a fixed u, ϵ'_k also decreases as k grows. Transmission, however, grows with both u (bigger digests) and k (more iterations). We write for the sake of clarity: $\epsilon'_k = \epsilon'_{k,u,\eta}$.

Fix η. Note that the number of bits transmitted per iteration ($\simeq 3ut$), is proportional to u. This yields an expected transmission complexity bound $T_{u,\eta}$ such that:

$$T_{u,\eta} \propto u\sum_{k=1}^{\infty} k\cdot\epsilon'_{k,u,\eta} = \frac{u\eta\,(\eta-1)}{2^u}\sum_{k=1}^{\infty} k\left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}}\right)^{k-1} = \frac{u\eta\,(\eta-1)\,8^u}{\left(2^u - 4^u + (\eta-2)^2\right)^2}$$

Dropping the proportionality factor $\eta\,(\eta-1)$, neglecting $2^u \ll 2^{2u}$ and approximating $(\eta - 2) \simeq \eta$, we can optimize the function:

$$\phi_\eta(u) = \frac{u\cdot 8^u}{(4^u - \eta^2)^2}$$

$\phi_{10^6}(u)$ admits an optimum for $u = 19$.

Note: The previous analysis is a rough approximation, in particular:

- We consider u-bit prime digests while u-bit strings contain only about $2^u/u$ primes.
- In all our probability calculations η can be replaced by the total number of differences t. It is reasonable to assume that in most *practical* settings $t \ll \eta$, but extreme instances where $t \sim \eta$ can sometimes be encountered as well.
- We used a fixed u in all rounds. Nothing forbids using a different u_k at each iteration[5].
- Our analysis treated t as a constant, but large t values increase p and hence the number of potential files detected as different per iteration - an effect disregarded in our analysis.

Given that, after all, optimization may only result in constant-factor improvements, we suggest to optimize t and u experimentally, *e.g.* using the open source program `btrsync` developed by the authors (*cf.* section 5).

[5] ...or even fine-tuning the u_ks adaptively, as a function of the laundry's effect on the progressively reconciliated multisets.

3.3 How to Stop a Probabilistic Washing Machine?

We now combine both optimizations and assume that ℓ laundry rounds are necessary for completing some given reconciliation task using a half-sized p. By opposition to section 2.2, confirming correct protocol termination is now non-trivial.

Let the round failure probability[6] be some function $\upsilon(w)$ (that we did not estimate). If w is kept small (for efficiency reasons), the probability $(1-\upsilon(w))^{\ell}$ that the protocol will properly terminate may dangerously drift away from one.

If v of $\ell+v$ rounds fail, Oscar needs to solve a problem called *Chinese Remaindering With Errors*:

Problem 1. (Chinese Remaindering With Errors Problem: CRWEP *[2]).* Given as input integers v, B and $\ell+v$ points $(s_1,p_1),\ldots,(s_{\ell+v},p_{\ell+v}) \in \mathbb{N}^2$ where the p_i's are coprime, output all numbers $0 \leq s < B$ such that $s \equiv s_i \bmod p_i$ for at least v values of i.

We refer the reader to [2] for more on this problem, which is beyond the scope of this article. Boneh [1] provides a polynomial-time algorithm for solving the CRWEP under certain conditions satisfied by our setting.

To detect that reconciliation succeeded, Neil will send to Oscar $\texttt{Hash}(\mathfrak{F}')$ as soon as the interaction starts. As long as Oscar's CRWEP resolution does not result in a state matching $\texttt{Hash}(\mathfrak{F}')$, the parties will continue the interaction.

4 Computational Complexity

Let $\mu(k)$ be the time required to multiply two k-bit numbers[7]. The modular division of two k-bit numbers and the reduction of $2k$-bit number modulo a k-bit number are known to cost $\tilde{O}(\mu(k))$ [1].

For naïve (*i.e.* convolutive) algorithms $\mu(k) = O(k^2)$, but using FFT multiplication strategies [5], $\mu(k) = \tilde{O}(k)$. FFT is experimentally faster than convolutive methods for $k \sim 10^6$ and on. For such sizes, in packages such as `gmp`, division and modular reduction also run in $\tilde{O}(\mu(k))$. Given that $p \sim 2^{ut}$, we get the following complexity analysis:

"reduction also run in $\tilde{O}(\mu(k))$"

Entity	Computation	Complexity expressed in $\tilde{O}$ of			
Both	redundancies c and c'	$n \cdot \mu(ut)$	naïve product	nut	using FFT
Oscar	$s = c'/c \bmod p$	$\mu(ut)$	naïve inversion	ut	using FFT
Oscar	a, b such that $s = a/b \bmod p$	$(ut)^2$	naïve ext. GCD	$\mu(ut)$	using [4,8]
Both	factorization of a (resp. b) by modular reductions	$n \cdot \mu(ut)$	naïve reduction	nut	using FFT
	Overwhelming complexity:	$\max((ut)^2, n \cdot \mu(ut))$			nut

Table 1. Global Protocol Complexity. In practice $\max((ut)^2, n{\cdot}\mu(ut)) = n(ut)^2$ because $(ut)^2 \ll n{\cdot}\mu(ut) = n(ut)^2$

[6] *i.e.* that probability that a round resulted in an $(a',b') \neq (a,b)$ satisfying equation (3).

[7] We suppose that $\forall k, k', \mu(k+k') \geq \mu(k) + \mu(k')$.

4.1 Improvements

The overwhelming complexities of the computations of (c, c') and of the factorizations can be reduced to $\tilde{O}(\frac{n}{t}\mu(ut))$ using convolutive methods and to $\tilde{O}(nu)$ with FFT [5]. To simplify explanations, assume that $t = 2^\tau$ is a power of two dividing n.

The idea is the following: group h_is by subsets of t elements and compute the product of each such subset in $\mathbb{N}$.

$$H_j = \prod_{i=jt}^{jt+t-1} h_i.$$

Each H_j can be computed in $\tilde{O}(\mu(ut))$ using the standard product tree method described in Algorithm 1 (for $j = 0$) and illustrated in Figure 1. And all these $\frac{n}{t}$ products can be computed in $\tilde{O}(\frac{n}{t}\mu(ut))$. Then, one can compute c by multiplying the H_j modulo p, which costs $\tilde{O}(\frac{n}{t}\mu(ut))$.

Algorithm 1 Product Tree Algorithm

Require: the set h_i
Ensure: $\pi = \pi_1 = \prod_0^{t-1} h_i$, and π_i for $i \in \{1, \ldots, 2t-1\}$ as in Figure 1

```
 1: π ← array of size t
 2: function prodTree(i,start,end)
 3:     if start = end then
 4:         return 1
 5:     else if start + 1 = end then
 6:         return h_start
 7:     else
 8:         mid ← ⌊(start+end)/2⌋
 9:         π_{2i} ←prodTree(2i,start,mid)
10:         π_{2i+1} ←prodTree(2i + 1,mid,end)
11:         return π_{2i} × π_{2i+1}
12: π_1 ←prodTree(1, 0, t)
```

The same technique applies to factorization[8], but with a slight *caveat*.

After computing the tree product, we can compute the residues of a modulo H_0. Then we can compute the residues of $a \mod H_0$ modulo the two children π_2 and π_3 of $H_0 = \pi_1$ in the product tree (depicted in Figure 1), and so on. Intuitively, we descend the product tree doing modulo reduction. At the end (*i.e.*, as we reach the leaves), we obtain the residues of a modulo each of the h_i ($i \in \{0, \ldots, t-1\}$). This is described in Algorithm 2 and illustrated in Figure 2. We can use the same method for the tree product associated to any H_j, and so we can get the residues of a modulo each of the h_i ($i \in \{jt, \ldots, jt+t-1\}$) for any j, *i.e.*, a modulo each of the h_i for any i. Complexity is $\tilde{O}(\mu(ut))$ for each j, summing-up to a total complexity of $\tilde{O}(\frac{n}{t}\tilde{O}(\mu(ut)))$.

[8] We explain the process with a, this is applicable *ne variatur* to b as well.

Algorithm 2 Division Using a Product Tree

Require: a an integer, π the product tree of Algorithm 1
Ensure: $A[i] = a \bmod \pi_i$ for $i \in \{1, \ldots, 2t-1\}$, computed as in Figure 2

```
1: A ← array of size t
2: function modTree(i)
3:     if i < 2t then
4:         A[i] ← A[⌊i/2⌋] mod π_i
5:         modTree(2i)
6:         modTree(2i + 1)
7: A[1] ← a mod π_1
8: modTree(2)
9: modTree(3)
```

4.2 Adapting *p*

Let `Prime`$[i]$ denote the i-th prime[9]. Besides conditions on size, the *only* property required from p is to be co-prime with all the h_i and all the h'_i. We can hence consider the following variants:

Variant 1: Use a sequence of smooth p_i:

$$p_i = \prod_{j=r_i}^{r_{i+1}-1} \texttt{Prime}[j]$$

Where the bounds r_i are chosen to ensure that each p_i has the proper size.

Variant 2: Define $p_i = \texttt{Prime}[i]^{r_i}$ where the exponents r_i are chosen to ensure that each p_i has the proper size.

Variant 3: Progressively work modulo a long power of two. This variant, is probably the most efficient of all but somewhat complex to explain. We hence describe it in detail in Appendix B.

This variant is more delicate to explain as Where the exponents r_i are chosen to ensure that each p_i has the proper size.

5 Implementation

We implemented and benchmarked the method that we describe in the previous sections. The implementation is called `btrsync`, its source code is available from [6].

The program implements unidirectional synchronization, which is simpler to understand. The implementation is divided in two subprograms: a shell script and a Python program:

5.1 The Shell Script

The shell script sets up two instances of the Python program on Oscar and Neil and establishes a bidirectional communication channel between them (materialized by two Unix pipes between their standard input and output).

[9] with `Prime`$[1] = 2$

5.2 The Python Program

The Python program uses gmp to perform all the number theory operations required, and performs the actual synchronization. It proceeds in two phases:

Finding Different Files

1. Compute the hashes of all files concatenated with their paths, type (folder/file), and permissions (not supported yet).
2. Implement the protocol proposed in Section **??** [add here a reference to the appropriate section in the paper] with input data coming from stdin and output data going to stdout.

More precisely: [TODO(antoine): isn't there going to be some overlap between this section and the description of the protocol in Appendix A? I feel we should explain the protocol elsewhere and only give implementation details here.]

- Oscar sends it product of hashes modulo a first prime number p_1.
- Neil receives the product, divides by its own product of hashes, reconstructs the fraction modulo p_1 [can we elaborate more on what happens here? which functions in GMP are used to do the reconstruction?] and checks if he can factor the denominator using his hashes base. If he can, he stops and sends the numerator and the list of tuples (path, type, hash of content of the file) corresponding to the denominator's factors. Otherwise he sends "None" [is this the ASCII string "None"? if not what does he send precisely?].
- If Neil sent "None", Oscar computes the product of hashes modulo another prime p_2, sends it... CRT mechanism... [can we elaborate more on what happens here? which functions in GMP are used to do the CRT?]
- If Neil sent the numerator and a list of tuples, then Oscar factors the numerator over his own hash values. Now each party (Neil, Oscar) knows precisely the list of files (path + type + hash of content) that differs from the over party.

[please structure the following:]

2. synchronize all the stuff [this is not an expression we can use in a paper...]. This part is not completely optimized.

We just remove all folders Oscar should not have and create new folders.

Then we remove all files Oscar should not have and synchronize using rsync the last files.

We could check for move (since we have the list of hash of contents of files) and do moves locally.

We can even try to detect moves of complete subtrees...

5.3 Move Resolution Algorithm

To reproduce the structure of Oscar on Neil, we have a list of file moves to apply. Sadly, it is not straightforward to apply the moves, because, if we take a file to move, its destination might be blocked, either because a file already exists (we want to move a to b, b already exists), or because a folder cannot be created (we want to move a to b/c, b already exists but is a file

and not a folder). Note that for a move operation $a \to b$, there is at most one file blocking the location b: we will call it the *blocker*.

If the blocker is not present on Oscar, then we can just delete it. However, if it exists, then we might need to move it somewhere else before we solve the move we are interested in. This move itself might have a blocker, and so on. It seems that we just need to continue until we reach a move which has no blocker or where the blocker can be deleted, but we can get caught in a cycle: if we must move a to b, b to c and c to a, then we will not be able to perform the operations without using a temporary location.

How can we perform the moves? A simple way would be to move each file to a unique temporary location and then rearrange files to our liking: however, this performs many unnecessary moves and could lead to problems if the program is interrupted. We can do something more clever by performing a decomposition in strongly connected components of the *move graph* (with one vertex per file and one edge per move operation going from to the file to its blocker or to its destination if no blocker exists). The computation of the SCC decomposition is simplified by the observation that because two files being moved to the same destination must be equal, we can only keep one arbitrary in-edge per node, and look at the graph pruned in this fashion: its nodes have in-degree at most one, so the strongly connected components are either single nodes or cycles. Once the SCC decomposition is known, the moves can be applied by applying each SCC in a bottom-up fashion, an SCC's moves being solved either trivially (for single files) or using one intermediate location (for cycles).

The detailed algorithm is implemented as two mutually recursive functions and presented as Algorithm 3.

An optimization implemented by `btrsync` over the algorithm described here is to move files instead of copying them and then removing the original file, because moves are faster than copies on most filesystems as the OS does not need to copy the actual file contents to perform them.

5.4 Experimental Comparison to `rsync`

We compared `rsync`[10] and our Divide & Factor implementation called `btrsync` under the following experimental conditions:

Test Directories: The directories used for transmission and time comparisons are described in Table 3.

Command-Line Options: `rsync` was called with the following options, for the reasons below:

▶ `--delete` to delete existing files on Oscar which do not exist on Neil like `btrsync` does.

▶ `-I` to ensure that `rsync` did not cheat by looking at file modification times (which `btrsync` does not do).

▶ `--chmod="a=rx,u+w"` in an attempt to disable the transfer of file permissions (which `btrsync` does not transfer). Although these settings ensure that `rsync` does not need to transfer

[10] `rsync` version 3.0.9, used both as a competitor to benchmark against and as an underlying call in our own code.

Algorithm 3 Perform Moves

Require: $\mathfrak{D}$ is a dictionary where $\mathfrak{D}[f]$ denotes the intended destinations of f

```
1: M ← []
2: T ← []
3: for f in 𝔇's keys do
4:     M[f] ← not_done
5: function UNBLOCK_COPY(f, t)
6:     if t is blocked by some b then
7:         if b is not in 𝔇's keys then
8:             unlink(b)                          ▷ We don't need b
9:         else
10:            RESOLVE(b)                         ▷ Take care of b and make it go away
11:    if T[f] was set then
12:        f ← T[f]
13:    copy(f, d)
14: function RESOLVE(f)
15:    if M[f] = done then
16:        return                                 ▷ Already managed by another in-edge
17:    if M[f] = doing then
18:        T[f] ← mktemp()
19:        move(f, T[f])
20:        M[f] ← done
21:        return                                 ▷ We found a loop, moved f out of the way
22:    M[f] ← doing
23:    for d ∈ 𝔇[f] do
24:        if d ≠ f then
25:            unblock_copy(f, d)                 ▷ Perform all the moves
26:    if f ∉ 𝔇[f] and T[f] was not set then
27:        unlink(f)
28:    if T[f] was set then
29:        unlink(T[f])
30: for f in 𝔇's keys do
31:    RESOLVE(f)
```

permissions, verbose logging suggests that it does transfer them anyway, so `rsync` must lose a few bytes per file as compared to `btrsync` for this reason.

▶ `-v` Transmission accounting was performed by calling `rsync` with the `-v` flag (which reports the number of sent and received bytes). For `btrsync` we added a piece of code counting the amount of data transmitted during `btrsync`'s own negotiations.

Network Configuration: Experiments were performed between two remote hosts connected by a high-speed link. Time measurements account both for CPU time and for transfer times and are hence only given as a general indication.

Results: Results are given in Table 2. In general, `btrsync` spent more time than `rsync` on computation (especially when the number of files is large, which is typically seen on the experiments involving `synthetic`). Transmission results, however, turn out to be favorable to `btrsync`.

In the trivial experiments where either Oscar or Neil have no data at all, `rsync` outperforms `btrsync`. This is especially visible when Neil has no data: `rsync` immediately notices that there is nothing to transfer, but `btrsync` engages in information transfers to determine the symmetric difference.

On non-trivial tasks, however, `btrsync` outperforms `rsync`. This is the case of the `synthetic` datasets, where `btrsync` does not have to transfer information about all unmodified files, and even more so in the case where there are no modifications at all. For Firefox source code datasets, `btrsync` saves a very small amount of bandwidth, presumably because of unmodified files. For the `btrsync` source code dataset, we notice that `btrsync`, unlike `rsync`, was able to detect the move and avoid retransferring the moved folder.

6 Conclusion and Further Improvements

Recopier de l'intro et adapter

We strongly encourage the developer community to continue improving our open source software (that we called `btrsync`).

7 Acknowledgment

The authors acknowledge Guillain Potron for his early involvement in this research work.

8 ToDo

– @Fabrice @David: Je n'ai toujours pas compris si on a un argument clair pour dire qu'on est meilleur que [4], du genre "on prend en compte la taille des hachés, etc." Ce serait à mon avis beaucoup plus vendeur que de dire "on fait pareil qu'eux, mais différemment" et de parler d'"interesting parameter trade-offs". :-)

- @David: Je ne comprends pas pourquoi tu parles de "constant-factor gains" par rapport à rsync. Vu que rsync ne détecte absolument pas les moves, on n'a pas juste un facteur constant comme amélioration, enfin si ton problème c'est "synchroniser un dataset de taille N avec un move d'un dossier de taille N/2" on a une complexité asymptotique largement meilleure que rsync, je crois ?
- Fix euclidean to Euclidean in reference 5.
- Merge two reference files rsynch and wagner.
- @Fabrice: Appendix B (je vais jeter un coup d'oeil la semaine prochaine)
- @Fabrice: Fig. 1 pas clair, je pense qu'il faut mieux créer une notation mu gcd car en fait la FFT ne modifie que le mu... Et on pourrait plutôt indiquer dans ce tableau la complexité (avec mu et mu gcd) en utilisant l'optimisation décrite ou sans utiliser l'optimisation décrite). N'oublions pas de préciser que les temps des expériences d'Antoine comptent aussi le tirage des nombres premiers (qui doit être négligeable peut-être dans notre cas, je ne me rappelle plus...)
- @Fabrice: Pour éviter le cas empty $\rightarrow$ source trop gros, on pourrait imaginer l'astuce suivante: si jamais Neil la taille de c est plus petite que la taille du produit des nombres premiers $p_1...p_n$ utilisés, Neil envoie un message pour l'indiquer, et on arrête là le protocole. Et Oscar peut directement factoriser ce nombre envoyé...
- @Fabrice: il faut discuter de la taille de la taille maximale des "petits" premiers utilisés pour les variantes de p et montrer que cela n'enlève pas trop d'entropie pour les h_i. Encore une fois, je m'en occupe la semaine prochaine si besoin.
- Revoir l'intro.

References

1. Burnikel, C., Ziegler, J., Im Stadtwald, D., et al.: Fast recursive division (1998)
2. Fouque, P.A., Stern, J., Wackers, J.G.: Cryptocomputing with rationals. In: Blaze, M. (ed.) Financial Cryptography. Lecture Notes in Computer Science, vol. 2357, pp. 136–146. Springer (2002)
3. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: Field, J., Hicks, M. (eds.) POPL. pp. 495–508. ACM (2012)
4. Pan, V., Wang, X.: On rational number reconstruction and approximation. SIAM Journal on Computing 33, 502 (2004)
5. Schönhage, A., Strassen, V.: Schnelle multiplikation grosser zahlen. Computing 7(3), 281–292 (1971)
6. Tridgell, A.: Efficient algorithms for sorting and synchronization. Ph.D. thesis, PhD thesis, The Australian National University (1999)
7. Vallée, B.: Gauss' algorithm revisited. J. Algorithms 12(4), 556–572 (1991)
8. Wang, X., Pan, V.: Acceleration of euclidean algorithm and rational number reconstruction. SIAM Journal on Computing 32(2), 548 (2003)

References

1. D. Boneh, *Finding Smooth Integers in Short Intervals Using CRT Decoding*, Proceedings of the 32-nd Annual ACM Symposium on Theory of Computing, 2000, pp. 265–272.
2. D. Bleichenbacher and Ph. Nguyen, Advances in Cryptology – Proceedings of Eurocrypt'00, vol. 1807 of Lecture Notes in Computer Science, Springer-Verlag, pp. 53–69.
3. Y. Minsky, A. Trachtenberg, *Scalable Set Reconciliation*, 40th Annual Allerton Conference on Communications, Control and Computing, Monticello, IL, October 2002. A full version entitled *Practical Set Reconciliation* can be downloaded from `http://ipsit.bu.edu/documents/BUTR2002-01.ps`

4. Y. Minsky, A. Trachtenberg, R. Zippel, *Set reconciliation with nearly optimal communication complexity*. IEEE Transactions on Information Theory, 49(9), 2003, pp. 2213–2218.
5. D. Eppstein, M. Goodrich, F. Uyeda, G. Varghese What's the difference?: efficient set reconciliation without prior context ACM SIGCOMM Computer Communication Review - SIGCOMM '11, 41(4), 2011, pp. 218-229.
6. `https://github.com/RobinMorisset/Btrsync`

A Extended Protocol

First phase during which Neil amasses modular information on the difference		
Oscar		**Neil**
		start the protocol with p_1
	$\xrightarrow{c_1}$	
		computes a, b using p_1
		if a factors properly then go to Final Phase else perform the protocol with p_2
	$\xrightarrow{c_2}$	
		computes $c \bmod p_1p_2 = \mathrm{CRT}_{p_1,p_2}(c_1, c_2)$
		computes a, b using p_1p_2
		if a factors properly then go to Final Phase else perform the protocol with p_3
	$\xrightarrow{c_3}$	
		computes $c \bmod p_1p_2p_3 = \mathrm{CRT}_{p_1,p_2,p_3}(c_1, c_2, c_3)$
		computes a, b using $p_1p_2p_3$
		if a factors properly then go to Final Phase else perform the protocol with p_4
	$\vdots$	
Final Phase		
		Let $\mathfrak{S} = \{F'_i \text{ s.t. } a \bmod h'_i = 0\}$
	$\xleftarrow{\mathfrak{S},b}$	
deletes files s.t. $b \bmod h_i = 0$ adds $\mathfrak{S}$ to the disk		

Note that parties do not need to store the p_i's in full. Indeed, the p_is could be subsequent primes sharing their most significant bits. This reduces storage per prime to a very small additive constant $\cong \ln(p_i) \cong \ln(2^{2tu+2}) \cong 1.39(tu+1)$ of about $\log_2(tu)$ bits.

B Power of Two Protocol

In this variant Oscar computes c in $\mathbb{N}$:

$$c = \prod_{F_i \in \mathfrak{F}} \texttt{HashPrime}(F_i) = \prod_{i=1}^{n} h_i \in \mathbb{N}$$

and considers $c = \bar{c}_{n-1}| \ldots |\bar{c}_2|\bar{c}_0$ as the concatenation of n successive u-bit strings.

First phase during which Neil amasses modular information on the difference	
Oscar	**Neil**
computes $c \in \mathbb{N}$	
$\xrightarrow{\bar{c}_0}$	
	computes a, b modulo 2^u if a factors properly then go to Final Phase else request next chunk $\bar{c}_1$
$\xrightarrow{\bar{c}_1}$	
	construct $c \bmod 2^{2u} = \bar{c}_1|\bar{c}_0$ computes a, b modulo 2^{2u} if a factors properly then go to Final Phase else request next chunk $\bar{c}_2$
$\xrightarrow{\bar{c}_2}$	
	construct $c \bmod 2^{3u} = \bar{c}_2|\bar{c}_1|\bar{c}_0$ computes a, b modulo 2^{3u} if a factors properly then go to Final Phase else request next chunk $\bar{c}_3$
$\vdots$ (for λ rounds)	

Final Phase	
	Let $\mathfrak{S} = \{F'_i \text{ s.t. } a \bmod 2^{\lambda u} = 0\}$
$\xleftarrow{\mathfrak{S},b}$	
deletes files s.t. $b \bmod 2^{\lambda u} = 0$ adds $\mathfrak{S}$ to the disk	

C Hashing Into Primes

Hashing into primes is frequently needed in cryptography. A recommended implementation of $\texttt{HashPrime}(F)$ is given in Algorithm 5. If u is large enough (*e.g.* 160) one might sacrifice uniformity to avoid repeated file hashings using $\texttt{HashPrime}(F) = \texttt{NextPrime}(\texttt{Hash}(F))$. Yet another acceleration option (that further destroys uniformity) consists in replacing `NextPrime` by the faster scanning Algorithm 4 where $\alpha = 2 \times 3 \times 5 \times \cdots \times \texttt{Prime}[d]$ is the product of the first primes until some rank d.

Algorithm 4 Fast Nonuniform Hashing Into Primes

$h = \alpha \left\lfloor \frac{\texttt{Hash}(F)}{\alpha} \right\rfloor + 1$
while h is composite **do**
    $h = h - \alpha$
return h

Algorithm 5 Possible Implementation of `HashPrime`(F)

$i = 0$
repeat
    $h = 2 \cdot \texttt{Hash}(F|i) + 1$
    $i = i + 1$
until h is prime
return h

Entities and Datasets		Transmission (Bytes)						Time (s)	
Neil's $\mathfrak{F}'$	Oscar's $\mathfrak{F}$	$\mathbf{TX}_r$	$\mathbf{RX}_r$	$\mathbf{TX}_b$	$\mathbf{RX}_b$	**abs**	**rel**	$\mathbf{t}_r$	$\mathbf{t}_b$
`source`	`empty`	1613	778353	1846	788357	10237	101 %	0.2	7.7
`empty`	`source`	11	29	12436	6120	18516	46390 %	0.1	5.5
`empty`	`empty`	11	29	19	28	7	118 %	0.1	0.3
`synthetic`	`synthetic_shuffled`	24891	51019	3638	4147	-68125	10 %	0.2	26.8
`synthetic_shuffled`	`synthetic`	24701	50625	3443	3477	-68406	9 %	0.2	26.6
`synthetic`	`synthetic`	25011	50918	327	28	-75574	0 %	0.1	25.7
`firefox-13.0.1`	`firefox-13.0`	90598	28003573	80895	27995969	-17307	100 %	2.6	4.2
`source_moved`	`source`	2456	694003	1603	1974	-692882	1 %	0.2	2.5

Table 2. Experimental results. The two first columns indicate the datasets, synchronization is performed *from* Neil *to* Oscar. RX and TX are received and sent byte counts, r and b are `rsync` and `btrsync`, we also provide the absolute difference in exchanged data abs $= \mathrm{TX}_b + \mathrm{RX}_b - \mathrm{TX}_r - \mathrm{RX}_r$ (positive when `btrsync` transfers more data than `rsync`) and the relative amount of data sent by `btrsync` compared to `rsync` rel $= (\mathrm{TX}_b + \mathrm{RX}_b)/(\mathrm{TX}_r + \mathrm{RX}_r)$(over 100% when `btrsync` transfers more data than `rsync`). The last two columns show timing results.

Directory	Description
`synthetic`	A directory containing 1000 very small files containing the numbers $1, 2, \ldots, 1000$.
`synthetic_shuffled`	`synthetic` with: 10 deleted files 10 renamed files 10 modified files
`source`	A snapshot of `btrsync`'s own source tree
`source_moved`	`source` with one big folder (a few megabits) renamed.
`firefox-13.0`	The source archive of Mozilla Firefox 13.0.
`firefox-13.0.1`	The source archive of Mozilla Firefox 13.0.1
`empty`	An empty folder.

Table 3. Test Directories.

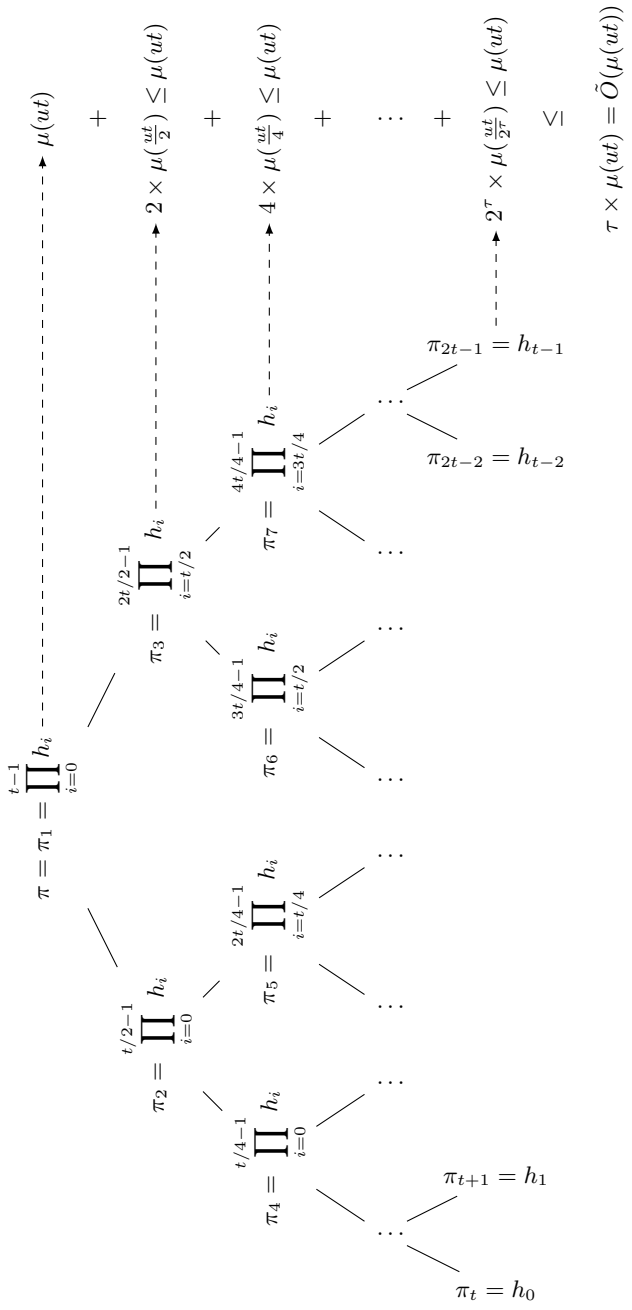

Fig. 1. Product Tree

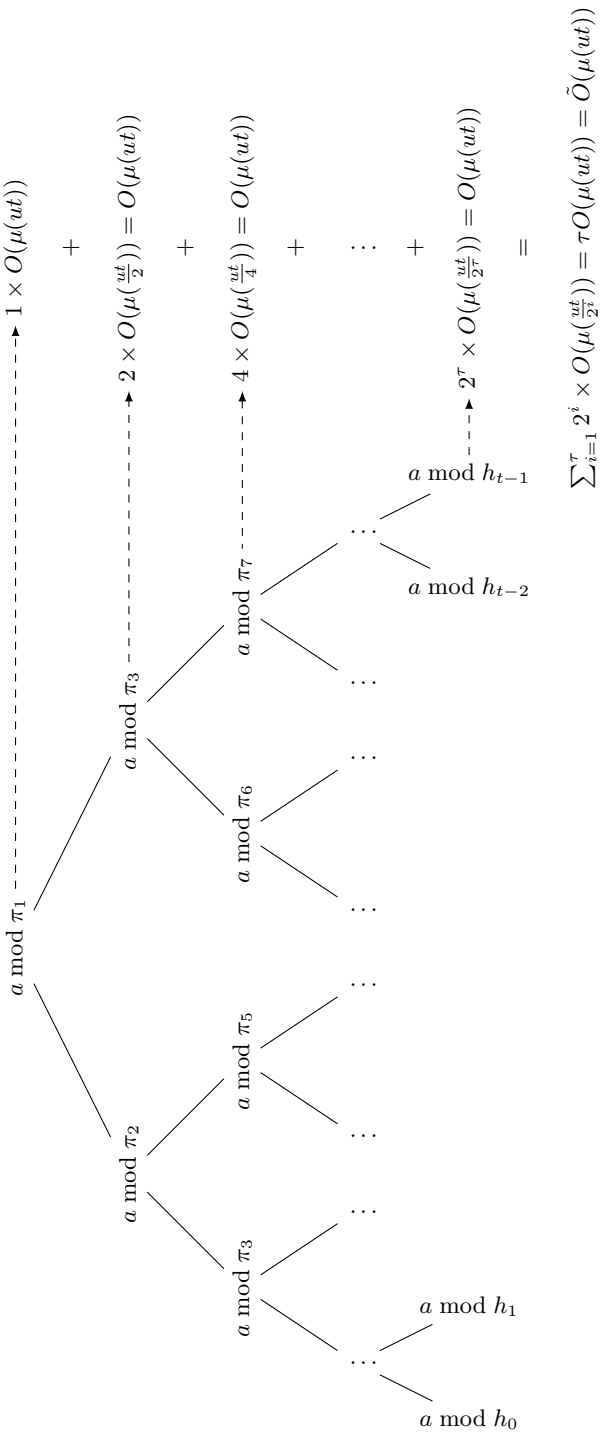

Fig. 2. Modular Reduction From Product Tree