Factor large numbers with parallelization

names...

December 22, 2016

1 Background

Being able to securely transer information is critial to communications, we rely on it everytime we go on internet where we want the information we send not to exposed to others. And this is made possible by encoding and deconding the message – suppose you want to send your account information to your bank, a *key pair* contains *public key* and *private key* will be created, another keypair will also be created in the bank,

$$\text{User}\begin{cases}\text{public key 1}\\ \text{private key 1}\end{cases} \quad \rightleftharpoons \quad \text{Bank}\begin{cases}\text{public key 2}\\ \text{private key 2}\end{cases}$$

you and bank will both make the public key open to everyone, so you can encode the message you want to send to the bank by using public key 2, and you know the only one can decode it is the bank with their private key 2. You will also encode your message with your private key 1 so that the bank will know it is truly you who sends the message by decoding it using your public key 1.

The encoding and decoding is done by using an algorithm called RSA, where it relies on the fact that it is extremely difficult to factor a large number to two prime numbers. We will first introduce a simple algorithm that is used to do prime factorization – Pollard's $p - 1$ Algorithm, and we will build a parallel version of this algorithm to see how much faster we can gain from the parallelization with a comparison to sequential implementation, then we will introduce a more sophisticate algorithm – Pollard Rho.

2 Euler-Fermat and Factoring

We will use the theorem of Fermat's little theorem(a special case of Euler-Fermat) to help us find the two prime factors of a given number, first we state an equivalent theorem[1],

Theorem 1. *If p is a prime number, then for any integer a, if a is not divisible by p, the number $a^{p-1} - 1$ is an integer multiple of p, i.e.*

$$a^{p-1} \equiv 1 \pmod{p}$$

Now suppose we have an integer $N = pq$ where p and q are prime, and from theorem 1 we have $a^{p-1} \equiv 1 \pmod{p}$ for a coprime with p, or equivalently $a^{p-1} = k_1 p + 1$ for some integer k_1,
Let L be an integer multiple of $p - 1$ i.e. $L = k_2(p - 1)$, then we have:

$$a^L = (a^{p-1})^{k_2} = (k_1 p + 1)^{k_2} = \binom{k}{0}1 + \binom{n}{1}(k_1 p) + \cdots + \binom{n}{n}(k_1 p)^{k_2}$$

so $a^L - 1$ is also a multiple of p, that is $a^L \equiv 1 \ (mod\ p)$.
Thus, greatest command divisor $gcd(a^L - 1, N) = p$ (for non-trivial solution e.g. 1 or N), so this gives us a way to find p(and q) if given $N = pq$, however, the problem is that we need to have $a^L - 1$ to do the calculation, and we do not know p yet how do we obtain L? We introduce *Pollard's $p - 1$ algorithm.*

2.1 Pollard's $p - 1$ algorithm[2]

We first start by 1) choosing a number a that is coprime to N, then we want to find a good integer $L = k_2(p-1)$, we do this by iterating over possible product of k_2 and $(p-1)$, more concretely, 2) let $L = k!$ with k iterate over $[1, 2, \cdots, \text{some large integer}]$, thus there is a good chance $L = k! = K_1 K_2$ will be equal to $(p - 1)k_2$ at some point of the iteration. Finally, we obtain p by 3) compute $gcd(a^L - 1, N)$, if the result is 1 or N then iterate to next k in step 2).

We show this by going through a real example,

Example: Factor 119

1) since 119 is odd, let us choose $a = 2$
2) $L = k!$, first we choose $k = 2$ so $L = 2$
3) compute gcd: $gcd(2^2 - 1, 119) = gcd(3, 119) = gcd(2, 3) = 1$
   this is a trivial solution, we change k to 4 in step 2)
   2) $L = 3! = 6$
   3) $gcd(2^6 - 1, 119) = gcd(63, 119) = gcd(56, 63) = gcd(7, 56) = 7$

so we have $p = 7$, this is correct as we used 7 × 17 to create the 119 in this example.
In the follwing sections we will discuss this algorithm's sequential and parallel implementation in Python.

2.2 Sequential implementation

the pseudo-code is straight froward according to example above, see complete code in appendix A.

```
Choose int a coprime to n
k = 1
L = k!
compute gcd(a^L-1, n)
if result is 1 or n:
    k += 1
recompute gcd(a^L-1, n)
repeat until non-trivial result obtained
```

Let us test it on $N = 3127$:

```
Finding a took: 0.0000078678 sec
Coprime a is:  2
Total time used 53.9340159893035889 sec
Prime factors are 53 ad 59
```

2.3 Parallel implementation

From *Example* we see there are two parts of the code one can paralleliaze, in part 1) we can use multi-processes to find the coprime a and 2) compute using different k at the same time, after running sequential code we see that part 2) takes the most computing time, thus we will try to make part 2) parallel.
The parallel jobs will be assigned to a cpu with 4 cores:

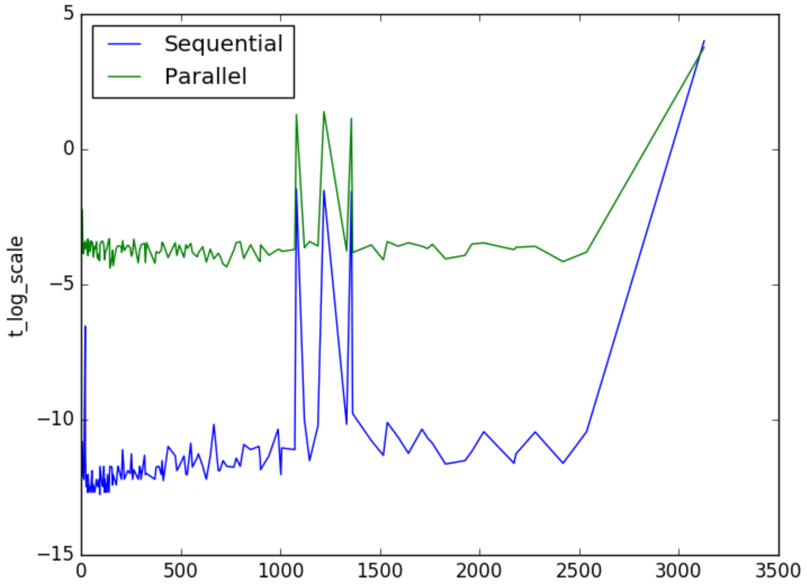
```
Choose int a coprime to n
k = 1
L = k!
compute gcd(a^k!-1, n)     -> p1
        gcd(a^(k+1)!-1, n) -> p2
        gcd(a^(k+2)!-1, n) -> p3
        gcd(a^(k+3)!-1, n) -> p4
if result is 1 or n:
    k += 4
recompute gcds
repeat until non-trivial result obtained
```

we expect this to produce result faster since four processes are running at the same time to find the correct factor, following is the result of running on $N = 3127$:

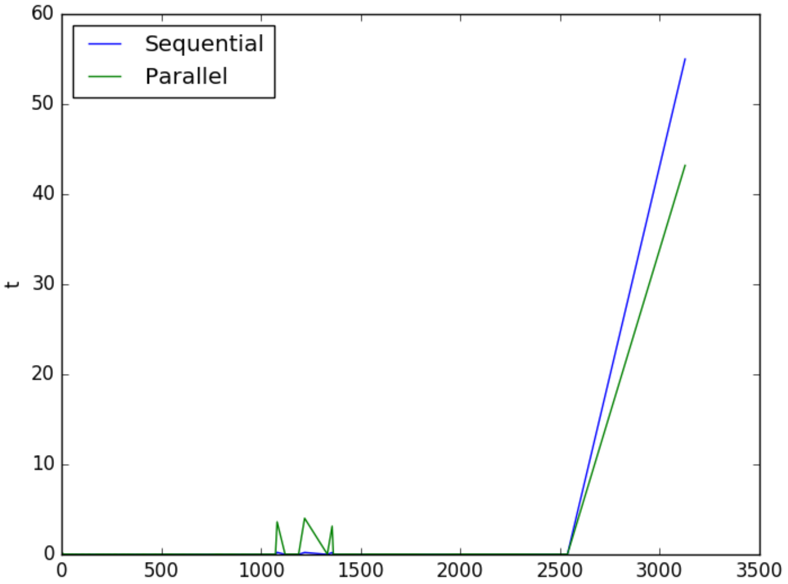
```
Finding a took: 0.0002229214 sec
Coprime a is:  2
Total time used 43.1968381404876709 sec
Prime factors are 53 ad 59
```

2.4 Comparison

From previous sections we see that for the number $N = 53 \times 59 = 3127$, parallel implementation is clearly faster than its sequential counterpart, however we will see that for number smaller than N sequential implementation would be faster, the reason is that for smaller number, the cost of overheads involved in parallelization outweighs its benefit, below we show the performance of sequential vs parallel, where x-axis is the number we want to prime factor and y-axis is time in seconds, first plot is in log-scale:



we can clearly see the presence of overheads when numbers are small, and the advantage of paralelization can be realized when N is large, e.g. $N = 53 \times 59$



So we see how parallelism can be used to accelerate the prime factor, and if the parallel is done on a GPU we could certainly expect a even shorter running time. In the following section, we will introduce a different prime factor technique called Pollard Rho, and from that we shall illustrate a point when applying parallelization that sometimes it may not be important how well the parallel program is written, but how efficient the base case – sequential implementation is.

3 Pollard Rho

...

Appendix A

...

References

[1] *Fermat's little theorem*,
`https://en.wikipedia.org/wiki/Fermat's_little_theorem`

[2] *pollards p 1 factorization*, Jeff Suzuki, Brooklyn College
`https://www.youtube.com/watch?v=fFJMoIj71nQ`