IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 19, Issue 2, Ver. V (Mar.-Apr. 2017), PP 51-54
www.iosrjournals.org

A Parallel Algorithm for Factorization of Big Odd Numbers

Dongbo FU
Department of Computer Science, Guangdong Neusoft Institue Foshan City, Guangdong Province, PRC, 528200

Abstract: *The article puts forward an algorithm to factorize a big odd number by means of subdividing the searching interval into finite independent subintervals. A divisor of a big odd number can be found in one of the subintervals. Owing to the independency of the subintervals, the algorithm can be performed in either sequential computing or parallel computing. Experiment shows that the algorithm is valid and practically applicable.*
Keywords: *Subdivision, Parallel, Searching Algorithm, Factorization*

I. Introduction

Factorization of integers has been an ancient hard problem in both mathematics and field of information system, as surveyed in article [1]. Article [2] combined the thoughts in articles [3] and [4] and put forward an algorithm that was declaimed to be almost as effective as that of Pollard's Rho. The algorithm that was introduced in article [2] first selects a mid-point that is proposed in article [3] and then select two intervals on both sides of the mid-point as objective searched intervals. This approach might be high effective when the odd number *N* that is going to be factorized is small. When *N* is really a very big number, the searched intervals will also be very big and it will still takes a very long time to find a divisor of *N*, especially when N is a semiprime. Hence, the approach in article [2] can be still improved. This article puts forward an improved one and introduces the details.

II. Definitions, Lemmas and Theorems

Lemmas mainly come from the theorems in articles [2] and [3].

Definition 1. An odd interval $[a,b]$ is a set of consecutive odd numbers that take *a* as lower bound and *b* as upper bound. For example, $[3,11]=\{3,5,7,9,11\}$.

Theorem 1. An odd interval $[a,b]$ contains $\frac{b-a}{2}+1$ consecutive odd numbers.

Proof. (Omitted)

Lemma 1. *Let* $m>2$ *be a positive integer and* $N=pq$ *be an odd composite number such that* $2^{m+1}+1\le N\le 2^{m+2}-1$, *where p and q are odd coprimed numbers that fit* $3\le p<q$ *; let* $e=2^{m+1}N-1$ *and* $l=\left\lfloor\frac{\sqrt{N}+1}{2}\right\rfloor$*; then in odd interval* $[e-2\times l,e]$ *there must exist an odd number* N_{mid} *that is a multiple of p and the bigger* $k=\frac{q}{p}$ *is the nearer* N_{odd} *is close to e. Distribution of* e, N_{mid} *and* $e-2\times l$ *can be illustrated in figure 1.*

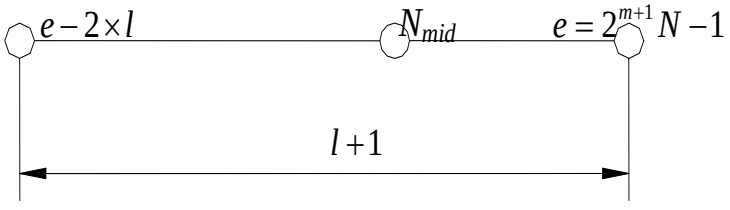

Fig.1 Distribution of Critical Nodes (m>2)

III. New Algorithm and Numerical Experiments

This section proposes an algorithm for factoring an odd composite number based on the theorems and corollaries introduced in the previous section. It first presents the thoughts of the algorithm design and then shows the new algorithm.

3.1 Principal Thought

Considering a search is performed on an odd interval $[e-2\times l, e]$, denoted by S_I; Referring to Theorem 1 and Lemma 1, S_I contains $l+1$ consecutive odd numbers among which there is a p's multiple N_{mid}. Therefore, it knows N_{mid} and N has a common divisor p and factorization of N turns to the problem to find common divisor between N and the odd number in S_I.

It knows that when N is a big number, the length of S_I is also very big. Hence subdividing S_I into small subintervals and searching in a parallel way on the small subintervals will be a natural choice. Then how to subdivide the interval S_I becomes a key issue. Let M_n^t (or *MNT*) be the *mean mount of numbers* that a computing cell can search per *unit time*; then the total time T_{total} to finish a number-by-number search is estimated theoretically by

$$T_{total} = \left\lfloor \frac{l+1}{2M_n^t} \right\rfloor = \left\lfloor \frac{\sqrt{N}+1}{4M_n^t} \right\rfloor$$

Obviously, an acceptable plan is that N_{mid} can be found in a *tolerable waiting time* τ. Hence a subdivision of S_I into $n_\tau = \frac{T_{total}}{\tau}$ subintervals will make each subinterval be searched in time τ. On the other hand, referring to conclusions in articles [3] and [4], it can infer that, not all the n_τ subintervals are required performing a number-by-number search because the p's multiple is near the middle of the interval S_I. Therefore, a search starts simultaneously from $e-2\times l$ rightwards and e leftwards in velocity M_n^t will be very close to N_{mid} by its both sides after a time t, as shown in figure 2.

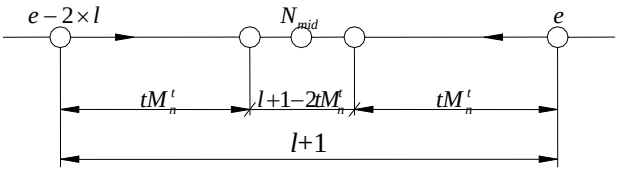

Fig. 2 A search close to the target

3.2 The Algorithm

Based on the thought of the algorithm design in previous section, an algorithm is designed to subdivide S_I into finite subintervals first and then to search from the mid-subinterval to its two-sided subintervals until the solution is obtained. The algorithm is as follows.

=== Subdivision & Mid-blossom Algorithm (SMA)=====

Input: Odd composite number N, M_n^t.

Step 1. Calculate the level on which S_I stays: $K = \lfloor \log_2 N \rfloor$;

Step 2. Calculate initial parameters:

$$l = \left\lfloor \frac{1}{2}(1+\sqrt{N}) \right\rfloor + 1$$

$$e_0 = 2^K N - 1;$$

$$Il = e_0 - 2\times l$$

Step 3. Calculate the following parameters.

(1) Numbers in S_I: $n_s = l$;

(2) n_{sn} and m_{sn} that satisfy

$$n_s = n_{sn}\cdot(2M_n^t) + m_{sn}, 0 \le m_{sn} < 2M_n^t,$$

$$\text{or } n_{sn} = \left\lfloor \frac{n_s}{2M_n^t} \right\rfloor,\ m_{sn} = n_s - (2M_n^t)\left\lfloor \frac{n_s}{2M_n^t} \right\rfloor;$$

Step 4. Subdivide S_I into $2n_{\tau n}+1$ subintervals by

$$S_I = s_{Il} \cup ... \cup s_{Il+n_{sn}-1} \cup s_{mid} \cup s_{Ir+n_{sn}-1} \cup ... \cup s_{Ir}$$

where

$s_{Il+i} = [Il + 2iM_n^t, Il + 2(i+1)M_n^t - 2], i = 0,1,...,n_{sn} - 1$

$s_{Ir+i} = [Ir - 2(i+1)M_n^t + 2, Ir - 2iM_n^t], i = 0,1,...,n_{sn} - 1$

$s_{mid} = [Il + 2n_{sn}M_n^t, Ir - 2n_{sn}M_n^t]$.

Step 5. Perform the following searches.

For every $e \in s_{mid}$, if(*FindGCD*(*N*, *e*)) return *GCD*;

For $i = n_{sn} - 1$ to $i = 0$

Begin

For every $e \in s_{Il+i}$, if(*FindGCD*(*N*, *e*)) return *GCD*;

For every $e \in s_{Ir+i}$, if(*FindGCD*(*N*, *e*)) return *GCD*;

End

==========End of Algorithm ==========

Remarks.

(1) The above SMA can be applied in parallel computations. Actually, if each subinterval is assigned to a computing cell of a parallel computing system, the parallel solution is very easy to execute. The algorithm can be even applied in a heterogeneous environment.

(2) The Step 5 can be alternatively performed by the following search.

==========An Alternative Perform==========

Step 5. Perform the following searches.

For $i = 0$ to $i = m_{sn} - 1$

Begin

$el = Il + 2(n_{sn} * M_n^t + i)$; if(*FindGCD*(*N*,*el*,)) return *GCD*;

End

For $i = 0$ to $i = M_n^t - 1$

For $j = 0$ to $j = n_{sn} - 1$

Begin

$el = Il + 2(j * M_n^t + i)$; if(*FindGCD*($N_{(0,0)}$,*el*)) return *GCD*;

$er = Ir - 2(j * M_n^t + i)$; if(*FindGCD*($N_{(0,0)}$,*er*)) return *GCD*;

End

==========

3.3 Numerical Experiments

Numerical experiments are made on a PC with an Intel Xeon E5450 CPU and 4GB memory via C++ gmp big number library. Experiment data from *N*1 to *N*10 originate from the article [3], N11 comes from article [5]. Tables 1 list the experimental results. In the table, the item *N's bits* means the number of N's decimal bits, the item *subs* means the number of total subintervals and the item *p's Loc* means the subinterval where *p*'s multiple lies.

Table 1 Experiment on Some Big Integers

Big Odd Number N	*N's bits*	*N's Factorization*	*M=1024*1024*	
			Subs	*p's Loc*
N1= 1123877887715932507	*19*	*299155897×3756830131*	*507*	*142*
N2=1129367102454866881	*19*	*25869889×43655660929*	*509*	*247*
N3=29742315699406748437	*20*	*372173423×79915205819*	*2063*	*1242*
N4=35249679931198483	*17*	*59138501×596052983*	*91*	*28*
N5=208127655734009353	*18*	*430470917×483488309*	*119*	*12*
N6=331432537700013787	*18*	*114098219×2904800273*	*277*	*111*
N7=3070282504055021789	*19*	*1436222173×2137748993*	*837*	*150*
N8=3757550627260778911	*19*	*16053127×234069700393*	*927*	*457*
N9=24928816998094684879	*20*	*347912923×71652460573*	*2383*	*1161*
N10=10188337563435517819	*20*	*70901851×143696355169*	*1525*	*744*
N11=1127451830576035879	*19*	*486100619×2319379541*	*509*	*231*

IV. Conclusion

Factorization of big integers usually involved vast computing cost. It is sure that, conventional computation can only fit for factoring small numbers. Thus using a parallel computing is regarded to be future trends in the computation. This article originates from such point of view. By subdividing the computing

interval into small ones, parallel computation can be surely applied on the computation. I am sure that, more better algorithm will come into being and it is not far from solving the problem of factoring a big integer.

Acknowledgements

The research work is supported by Foshan Bureau of Science and Technology under projects 2016AG100652, 2016AG100792 and 2016AG100382. The author sincerely presents thanks to them all.

References

[1]. Sonal Sarnaik, Dinesh Gadekarand Umesh Gaikwad, An overview to Integer factorization and RSA in Cryptography, *International Journal for Advance Research in Engineering and technoloy,2014,2(9):21-26*

[2]. Jianhui LI, Algorithm Design and Implementation for a Mathematical Model of Factoring Integers,*IOSR Journal of Mathematics,2017,13(I Ver. VI):37-41*

[3]. Aldrin W, Wanambisi Shem Aywa, Cleophas Maende, etc, Factorization of Large Integers, *International Journal of Mathematics and Statistics Studies,2013,1(1):39-44*

[4]. Xingbo WANG, Genetic Traits of Odd Numbers With Applications in Factorization of Integers, *Global Journal of Pure and Applied Mathematics,2017,13(2) 493-517*

[5]. Ulrich H Kurziweg,Factoring Large Composite Numbers, *http://www2.mae.ufl.edu/~uhk/ FACTORING-LARGE-COMPOSITE-NUMBERS.pdf*