社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2004－ＣＳＥＣ－26　(13)
2004／7／20

一般数体篩法実装実験（5）
—バケツ整列を利用した篩の高速化

青木和麻呂†　　植田　広樹†

† NTT
〒 239-0847 神奈川県横須賀市光の丘 1-1
E-mail: †{maro,ueda}@isl.ntt.co.jp

あらまし　数体篩法などの篩アルゴリズムにバケツ整列を利用した高速化法を提案する．篩処理では大量のメモリ更新が行なわれるが多くの場合キャッシュミスが起きる．提案手法を用いると劇的にキャッシュミスが減る．この方法は，篩領域がキャッシュ領域の平方ぐらいまで有効で，単純な実装法の数倍は高速になる．
キーワード　素因数分解，篩，数体篩法，largish prime，バケツ整列，基数整列

A trial of GNFS implementation (Part V)
—Speeding up sieving using bucket sort

Kazumaro AOKI† and Hiroki UEDA†

† NTT
1-1 Hikarinooka, Yokosuka-shi, Kanagawa-ken, 239-0847 Japan
E-mail: †{maro,ueda}@isl.ntt.co.jp

Abstract　This paper proposes a new sieving algorithm that employs a bucket sort as a part of a factoring algorithm such as the number field sieve. The sieving step requires an enormous number of memory updates; however, these updates usually cause cache hit misses. The proposed algorithm dramatically reduces the number of cache hit misses when the size of the sieving region is roughly less than the square of the cache size, and the memory updates are several times faster than the straightforward implementation.
Key words　integer factoring, sieving algorithm, number field sieve, largish prime, bucket sort, radix sort

1. はじめに

素因数分解問題は，公開鍵暗号として，安全性がその困難性に基づく RSA 暗号がもっとも広く使われていることから，興味深い問題と考えられている．現在，RSA の法に用いられるような合成数の分解において，数百ビット以上の場合は数体篩法 [1] が最も高速である．

数体篩法は，多くの処理から構成される．そのうち，篩処理は理論的にも実験的にも最も困難であることが知られている．PC 上の愚直な篩の実装ではメモリの読み書きに非常に時間を要する．もし，全てのメモリ読み書きがキャッシュに当たるのだとすると数十倍は篩処理が高速になる．

本稿では，PC 上の篩処理の実装において，メモリの読み書きの高速化に焦点を当て，メモリの読み書きでキャッシュに当たる可能性を高める方法を提案する．実験により，提案法を用いると愚直な実装法に比べ篩処理が数倍高速になることを確認した．

2. 準　　備

2.1 数体篩法

この説では数体篩法について以下で用いる記号などを簡単に説明する．詳細については文献 [1] などを参照されたい．

N を分解したい合成数とする．まず，既約多項式 $f(X) \in \mathbf{Z}[X]$ と，$f(M) \equiv 0 \pmod{N}$ を満たす，根 M を見つける．篩処理の目的は $N_R(a,b) = |a + bM|$ が B_R-smooth であり，

(注0)：本研究の一部は通信・放送機構の委託研究及び CRYPTREC の支援を受けた

[Algorithm 1] [line sieve for rational side (基本版)]

1: **for** $b \leftarrow 1$ **to** H_b
2: $\quad$ S[a] を $\log N_R(a,b)$ に初期化 ($-H_a \leqq \forall a < H_a$)
3: $\quad$ **for prime** $p \leftarrow 2$ **to** B_R
4: $\quad\quad$ b と p から初期篩点 $a \geqq -H_a$ を計算
5: $\quad\quad$ **while** $a < H_a$
6: $\quad\quad\quad$ S[a] $\leftarrow$ S[a] $- \log p$
7: $\quad\quad\quad$ $a \leftarrow a + p$
8: $\quad$ S[a] が小さい全ての a に対し $N_R(a,b)$ を素因数分解

$N_A(a,b) = |(-b)^{\deg f} f(-a/b)|$ が B_A-smooth(注1) となる互いに素な $(a,b) \in \mathbf{Z}^2$ をたくさん見つけることである．このような組 (a,b) を relation と呼ぶ．

line-by-line sieve (以下では単に line sieve と呼ぶ) を Algorithm 1 として示す．このアルゴリズムは relation を見つけるための最も基本的なアルゴリズムである．今後，代数側について有理側とほとんど同じなので省略する．Algorithm 1 では事前に $2H_a$ 個の要素が配列 S として確保されていることを仮定している．もし，$2H_a$ が実装環境において巨大であれば，この篩領域は分割してもよい．多くの場合 S[a] は 1 バイトであり，$\log p$ の底は S[a] で表すことが出来る最大値を越えないよう定められる．Step 8 の「小さい」ことを定める閾値は Step 2 や 6 で対数を整数に丸めた際に生じる誤差や，素数冪に対する篩処理(注2) の省略から生じる誤差を考慮して定められる．

2.2 large prime variation

もし，B_R が H_a 程度以上ならば，Step 5 は滅多に実行されず Step 4 の初期篩点の計算が篩時間の大半を占める可能性がある．このような場合 large prime variation という方法が考えられている．基本版からの変更点は次の通りである．

(1) Step 3 の p の上限を B_R^L ($< B_R$) に

(2) Step 8 の閾値を緩める

B_R^L 以上の小さな数に対する高速な素数判定や素因数分解が可能になればなるほど閾値を緩めることが出来る．

経験によると，large prime variation の最も重い計算は Step 6 で行なわれる S[a] のメモリ更新である．本稿では，このメモリ更新を最適化する．

2.3 PC のメモリ待ち時間

最近の PC はキャッシュメモリ (以下単にキャッシュ) を備え，キャッシュは多くの場合，複数のレベルからなっている．数字が少ないレベルのキャッシュは高速かつ小容量である．表 1 に Pentium 4 の一般レジスタで論理演算を行なう場合のメモリ特性を例としてあげる．

PC のメモリは連続する番地への読み書きが高速になるように設計されており，逆にランダムな番地への読み書きの性能は非常に貧弱である．Algorithm 1 の Step 6 にあるように

(注1)：「x が y-smooth」とは x の全ての素因子が y 以下という意味である．
(注2)：素数冪 p^e を素数，$\log p^e$ を $\log p$ とみなすことにより，素数冪も Algorithm 1 に簡単に取り込める．

表 1 Pentium 4 Northwood [2, p.1-17,1-19,1-20]

	line size	容量	latency
レジスタ	(4 B)	32 B	$\frac{1}{2}$ processor cycle
レベル 1 キャッシュ	64 B	8 KB	2 processor cycles
レベル 2 キャッシュ	64 B+64 B	512 KB	7 processor cycles
主記憶	(4 KB)	≈1 GB	12 processor cycles + 6-12 bus cycles

[Algorithm 2] [line sieve から block sieving への追加部]

1: **for** $a^S \leftarrow -H_a$ **to** H_a **step** $+H_a^S$
2: $\quad$ **for prime** $p \leftarrow 2$ **to** B_R^S
3: $\quad$ 初期篩点 $a \geqq a^S$ を計算
4: $\quad$ **while** $a < a^S + H_a^S$
5: $\quad\quad$ S[a] $\leftarrow$ S[a] $- \log p$
6: $\quad\quad$ $a \leftarrow a + p$

line sieve は p 間隔で S[a] を更新する．p がキャッシュより大きい時は主記憶にとって S[a] の更新はランダムアクセスに見える．主記憶からの読みだしは表 1 によると Pentium 4 の動作周波数が 2.53 GHz で FSB が 533 MHz の場合は少なくとも $12 + 6 \times (2.53/0.533) = 40.5$ processor cycles を要する．しかしながら，実際に必要となる主記憶からのメモリ読み時間はさらにかかることが感じられるだろう．実際，簡単なランダムメモリ読みだし実験によると主記憶からの読みだしには数百 processor cycles を要することが分かる．

2.4 これまでの結果

Algorithm 1 の Step 5 から 7 を見れば分かるように篩処理の最内周ループは小さく非常に単純であるので，ほとんどがメモリ待ちと考えることが出来る．このキャッシュミスに対抗するため，[3] は block sieving 法を提案している．基本的な line sieve である Algorithm 1 と block sieving 法の違いは次の 2 点である．

- Step 2 と 3 の間に Algorithm 2 を導入
- Step 3 の p の初期値を B_R^S 以上の最小の素数に修正

block sieving 法は因子基底の素数を smallish prime ($\in (0, B^S]$) と largish prime ($\in (B^S, B^L]$) に分類し，幅 H_a^S の小さな区間毎に smallish prime でメモリを更新する．最大限の性能を引き出すためには H_a^S と B_R^S をおよそキャッシュの大きさに設定するのがよい．初期篩点の計算は Step 4 で計算した最終篩点を取っておけば省略できる．メモリの階層構造に注目し，smallish prime をさらに細かく分類するとよりよい性能が得られる可能性がある．

3. バケツ整列を利用した篩

2.4 節で紹介した block sieving 法を用いると smallish prime に対するキャッシュミスは激減するが，largish prime に対する篩処理はまだ多くのキャッシュミスを引き起こす．この章では，largish prime の篩処理で発生するキャッシュミス回数をバケツ整列 [4, Section 5.2.5] を利用して減らす方法を説明する．

2.3 節で説明したように近い番地に対するメモリ更新

[Algorithm 3]

1: すべてのバケツを空にする
2: **for prime** $p \leftarrow B_R^S + 1$ **to** B_R^L
3: 初期篩点 $a \geqq -H_a$ を求める
4: **while** $a < H_a$
5: $\left\lfloor \frac{a+H_a}{r} \right\rfloor$ 番目のバケツに $(a, \log p)$ を入れる
6: $a \leftarrow a + p$

[Algorithm 4]

1: **for** i 番目のバケツ $(0 \leqq i < n)$ に対し
2: **for** i 番目のバケツに入っている全ての $(a, \log p)$ に対し
3: $\mathrm{S}[a] \leftarrow \mathrm{S}[a] - \log p$

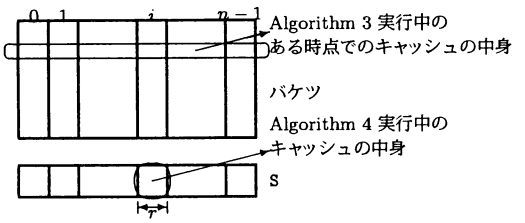

図 1 バケツと S のメモリ利用状況

はキャッシュに当たり，また $\log p$ を減じる処理は可換である．もし $(a, \log p)$ が a を鍵として整列されれば，キャッシュミスはほとんどなくなる．但し，$\mathrm{S}[a]$ の更新回数は大体 $2H_a(\log\log B^L - \log\log B^S)$ つまり H_a に対しほとんど線形なので，整列処理は非常に高速でなければならない．しかしながら完全な整列は必要とされていないこと，最近の PC は非常に多くの主記憶を備えていることを利用し，バケツ整列を使ってこの問題を解決する．

3.1 提 案 法

提案法は Algorithm 1 の largish prime に対する篩処理の置換えである．つまり，largish prime に対する Algorithm 1 の篩処理と機能は同じである．

この方法はバケツ整列に基づいている．n をバケツの数，n_S を S の要素数，r を $\left\lceil \frac{n_S}{n} \right\rceil$ とする．つまり Algorithm 1 においては $n_S = 2H_a$ である．提案法は，Algorithm 3 と Algorithm 4 の連続実行からなる．Algorithm 3 は $(a, \log p)$ をバケツに投入し，Algorithm 4 はバケツの要素を用い $\mathrm{S}[a]$ を更新する．

3.2 なぜ提案法はキャッシュミスを起こさないか

図 1 を使い説明する．まず Algorithm 4 を考える．あるバケツの全ての要素は範囲 r のメモリしか更新しない．つまり

$$r \times (\text{それぞれの } \mathrm{S}[a] \text{ の大きさ}) \leqq (\text{キャッシュの大きさ}) \quad (1)$$

でなければならない．次に Algorithm 3 を考える．それぞれのバケツについて，メモリへの書き込み番地は連続している．つまり

$$n \times (\text{cache line の大きさ}) \leqq (\text{キャッシュの大きさ}) \quad (2)$$

が成り立てば十分である(注3)．ここで，$(a, \log p)$ の大きさは cache line の大きさよりは小さいとも仮定している．式 (1) と (2) を合わせると

$$n_S \times (\text{それぞれの } \mathrm{S}[a] \text{ の大きさ}) \leqq \frac{(\text{キャッシュの大きさ})^2}{(\text{cache line の大きさ})} \quad (3)$$

が成り立てば n が存在する．

表 1 を用い，典型的なパラメータを考えよう．キャッシュの大きさは 512 KB であり，cache line の大きさは 128 B である．従って，不等式 (3) の右辺は 2^{31} B となる．もし，それぞれの $\mathrm{S}[a]$ に対し 1 B を割り付けるなら，S は 2 GB を占めることになる．つまり，提案法は多くの PC に対し有効である．提案法は，メモリの読み書きを増加させるが，適切な先読みを行なうことによりキャッシュミス回数を劇的に減らす．

3.3 最適化と改良

この節では，提案法の最適化と改良について紹介する．

3.3.1 $(a, \log p)$ の大きさの削減

i 番目のバケツでは，$a' = a + H_a \bmod r$ だけ保存すれば，$a = ir + a' - H_a$ であるので，a を復元できる．よって，バケツ中の a の占めるビット数は削減できる．

さらに $(a, \log p)$ の生成は p について昇順であるように出来るので，バケツ中の $\log p$ の値は前の値に比べてほとんど同じかまたは +1 であると考えられる．つまり $\log p$ のビット数は 1 ビットにすることが出来る．

3.3.2 バケツの数

Algorithm 3 の Step 5 や 3.3.1 節の方法を効率的に実行するためには，ほとんどの PC では r は 2 の冪にする必要があるだろう．

3.3.3 階層的なバケツ

基数整列の考え方やキャッシュの階層構造を考慮すると，より小さなバケツを使い Algorithm 4 を Algorithm 3 と Algorithm 4 に置き換えるとより効率がよい場合がある．

3.3.4 バケツ用のメモリ利用量の削減

PC の主記憶に全ての $(a, \log p)$ は格納できない状況を考える．この場合は，Algorithm 3 の Step 5 でバケツがいっぱいになるたびに Algorithm 4 を呼び出し，バケツを空にするとよい．

3.3.5 篩メモリ S の削減

まず，Algorithm 3 と 4 を用い largish prime に対する篩を実行する．Algorithm 4 の実行中の Step 1 と 2 の間に smallish prime の篩を行なう．i 番目のバケツについて a は $[ir - H_a, (i+1)r - H_a)$ に入っているので，$\mathrm{S}[a]$ の大きさとしては，r 個分準備すれば十分である．

残念ながらこの方法は 3.3.4 節で紹介した方法と同時には使えないことに注意せよ．

3.3.6 試し除算へのバケツの再利用

Algorithm 1 の Step 8 の実行時間を削減するために試し除算法 [5] が提案されている．この方法は，ほとんどここまで議論

(注3)：ここでキャッシュは cache line と呼ばれる単位でしか更新できないことに注意せよ．

表 2　因子基底の上限と対応するアルゴリズム

範囲	名称	アルゴリズム
$p \leq B^T$	p: tiny prime	篩パターン
$B^T < p \leq B^S$	p: smallish prime	block sieving
$B^S < p \leq B^L$	p: largish prime	バケツ整列
$B^L < p \leq B$	p: large prime	素数判定と分解

してきた篩と同じであるが少数の (a, b) に対してのみ篩を実行する点が異なる．Algorithm 3 でバケツを満たす際，$(a, \log p)$ の他に p も保存すればバケツの中身を試し除算で再利用できる．この方法により試し除算で largish prime に対する初期化点の計算などを省略できる．しかしながら，p を保存するということは，バケツ用のメモリが倍は必要になるだろう．この場合，全ての p を保存するのではなく，largish prime のうちでもある程度小さな素数のみをバケツに保存するという手法も考えられる．

3.3.7　lattice sieve への適用

提案法はメモリ更新操作が可換であればどのようなアルゴリズムにも適用できる．従って，lattice sieve に提案法を用いることには何の問題もない．

3.3.8　tiny prime に対する篩パターン

[6, p.334] は S[a] を tiny prime の log で初期化することを提案している．これは次の方法で効率的に実現できる．最初に，tiny prime (B^T 以下の素数) とその冪で，それらの最小公倍数で決定される篩パターンを計算する．一旦，この篩パターンを計算すれば，最初の場所の調整さえすればこのパターンの使い回しにより S[a] の初期化ができる．

3.4　関連研究

我々とは独立に [7] で $(a, \log p)$ を整列させる篩ハードウェアが提案されている．この方法では，キャッシュについては考慮していないが，篩処理を整列とみなせる点が似ている．

4.　Pentium 4 上の実装

lattice sieve において Algorithm 3 と Algorithm 4 を 1 GB 主記憶，533 MHz FSB の Pentium 4 (Northwood) に実装した．ここで，3.3.4 節また 3.3.1 節の $\log p$ のビット数削減(注4)以外の全ての 3.3 節にある改良は導入した．PC の特性は表 1 の通りである．因子基底の上限は表 2 に示した．これら分類された素数の名前は [8] に近いものがあったので，それを流用した．これからの因子基底の上限 B の最適値を見つけるために [9] にある c158 にあるパラメータを利用した．

4.1　パラメータ選択

$\log p$ と $(a, \log p)$ の大きさについて，Pentium 4 の基本的なデータは 4 B であること，メモリ読み書きの最小単位は 1 B であることを考慮し，それぞれ 1 B，4 B 割り当てた．

c158 の分解において，篩領域は $2H_c \times H_d = 2^{14} \times 2^{13}$ であった．この矩形領域を line sieve の場合に翻訳すると $2H_a = 2^{14} \times 2^{13} = 2^{27}$ となる．それぞれの relation に含まれる最大 large prime の数，及び B_R^L と B_A^L について情報がなかったので，large prime の積に対する我々の素数判定，分解プログラム，また line sieve での因子基底の上限やレベル 2 のキャッシュの大きさを考慮し，最大 large prime 数を algebraic 側，rational 側ともに 2 個，因子基底の分類を決める上限をそれぞれ $B_R^L = 30 \times 10^6$，$B_A^L = 0.9 \times Q$，$B_R^S = B_A^S = 512 \times 2^{10}$ と取った．ここで，Q は special-Q(注5) の Q である．上記パラメータを用い，深さ 1，2，3 の階層バケツについて全てのあり得る 2 冪となる r の組合せについて実験を行なった．その結果，最適な階層の深さは 2 であった．予想に反し最適な r の組合せは，レベル 2 とレベル 1 のキャッシュの大きさの組合せではなく，2 MB と 256 KB に取った場合であった．

次に，B_R^S と B_A^S の最適値を探した．いくつかの実験の結果，$B_R^S = 2H_c$ および $B_A^S = 5H_c$ と取った場合に，ほぼ最適であることが分かった．

注 1　素数冪については冪の値が $\sqrt{B^L}$ 以下のものについてのみ篩を行なった．また $B_R^T = B_A^T = 5$ と取った．

注 2　smallish prime について，キャッシュの大きさおよび篩範囲を考慮し，さらに細かく分類して block sieving を行なった．

注 3　Algorithm 3 実行後，それぞれのバケツにはいっている要素の数はほぼ同じであった．我々が行なった実験では 2%の差が最大であった．

注 4　large prime の積に対する素数判定には 2 を底とする Solovay-Strassen 法 [10, pp.90–91]，素因数分解には ρ 法 [10, pp.177–183] と SQUFOF 法 [10, pp.186–193] を用いた [11].

4.2　分解例

これまでに説明した方法を実装し，$2^{1826}+1$ の 164 桁因子である c164 を GNFS により分解した．分解の全体の概要は [12] を参照のこと．c164 分解に用いたパラメータを表 3 にまとめた．参考のため，表 3 には RSA-512 の分解で用いられたデータ [13] も併記した．

c164 は RSA-512 よりも大きいにも関わらず，提案法を用いて実装した篩は MIPS 換算で同じ計算量あたり多くの relation を生成した．但し，MIPS は篩の計算量比較のためには適していないこと，分解に用いた計算機の特性は異なることから，単純な比較は避けなければならない．

注 1　RSA-512 で用いられた lattice sieve 実装は RSA-130 の分解のために作られた [13, Section 3.2] ものである．

注 2　我々の篩の計算量算出のために使った MIPS は「BYTE benchmark」の出力により算出した．「Dhrystone 2 without register variables」が 3969679.6 lps であったので，

(注4)：本稿投稿直前に思いついたので実装が間に合っていない．

(注5)：lattice sieve では，$N_A(a, b)$ があらかじめ定めた素数で割れる (a, b) のみを篩対象とすることを繰り返す．その，あらかじめ定めた素数のことを special-Q と呼ぶ．

表 3 lattice sieve で用いられたパラメータ

	H_c	H_d	B_R^L	B_A^L	B	max sp-Q	#sp-Q	#LP	rel/MY
c164	16 K	8 K	40 m	$0.95Q$	4 g	194 m	8.2 m	2+2	29 k
RSA-512	4 K	5 k	16 M	16 M	1 g	15.7 m	308 m	2+2	14 k

k: 10^3, K: 2^{10}, m: 10^6, M: 2^{20} g: 10^9, G: 2^{30}

rel/MY: 1 MIPS year 当たり生成された relation 数

MIPS は 3969679.6/1767 ≈ 2246.6 と算出された．この数字を表 3 の c164 の rel/MY 算出に用いた．

注 3 RSA-576 が GNFS により既に分解されており [14]，さらにその時使われた篩は RSA-512 に使われた篩より高速に見えるが，公開されている情報だけでは計算量を見積もるのに不十分であるので，十分に情報が開示されていてかつ最も大きな合成数の分解結果である [13] と比較した．

5. ま と め

本稿でキャッシュメモリを賢く用いる新たな篩アルゴリズムを提案した．このアルゴリズムを用いると，篩処理でのメモリ更新は単純な $\log p$ の減算と比較して数倍は高速になる．さらに，我々はこのアルゴリズムを lattice sieve に適用し Pentium 4 上に実装した．その結果，164 桁の合成数を一般数体篩法により分解できた．

謝辞 本稿に関し有益な議論，特に 3.3.5 節および 3.3.6 節後半のアイディアを出して下さった上に 4.1 節の B^S の推定をして下さった立教大学の木田祐司先生，3.3.1 節の $\log p$ のビット数削減の示唆をして頂いた富士通研究所の下山武司博士及び日立製作所の古屋聡一博士に大変感謝します．

文 献

[1] A. K. Lenstra and H. W. Lenstra, Jr. Eds.: "The development of the number field sieve", Vol. 1554 of Lecture Notes in Mathematics, Springer-Verlag, Berlin, Heidelberg (1993).

[2] Intel Corporation: "IA-32 Intel Architecture Optimization Reference Manual" (2004). Order Number: 248966-010 (http://support.intel.com/design/pentium4/manuals/248966.htm).

[3] G. Wambach and H. Wettig: "Block sieving algorithms", Technical Report 190, Informatik, Universität zu Köln (1995). (http://www.zaik.uni-koeln.de/~paper/index.html?show=zpr95-190).

[4] D. E. Knuth: "Sorting and Searching", Vol. 3 of The Art of Computer Programming, Addison Wesley, second edition (1998).

[5] R. A. Golliver, A. K. Lenstra and K. S. McCurley: "Lattice sieving and trial division", Algorithmic Number Theory — First International Symposium, ANTS-I (Eds. by L. M. Adleman and M.-D. Huang), Vol. 877 of Lecture Notes in Computer Science, Berlin, Heidelberg, New York, Springer-Verlag, pp. 18–27 (1994).

[6] R. D. Silverman: "The multiple polynomial quadratic sieve", Mathematics of Computation, **48**, 177, pp. 329–339 (1987).

[7] W. Geiselmann and R. Steinwandt: "A dedicated sieving hardware", Public Key Cryptography — PKC2003 (Ed. by Y. G. Desmedt). Vol. 2567 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, New York, pp. 254–266 (2003).

[8] A. Shamir and E. Tromer: "Factoring large numbers with the TWIRL device", Advances in Cryptology — CRYPTO 2003 (Ed. by D. Boneh), Vol. 2729 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, New York, pp. 1–26 (2003).

[9] F. Bahr, J. Franke and T. Kleinjung: "Factorization of 158-digit cofactor of $2^{953}+1$", (available at http://www.crypto-world.com/announcements/c158.txt) (2002).

[10] H. Riesel: "Prime Numbers and Computer Methods for Factorization", Vol. 126 of Progress in Mathematics, Birkhäuser, Boston, Basel, Berlin, second edition (1993).

[11] K. Aoki, T. Izu, H. Ueda and T. Shimoyama: "A trial of GNFS implementation (part II) —mini-factoring & mini-primarity test", Technical Report IT2003-112, ISEC2003-152, WBS2003-230, The Institute of Electronics, Information and Communication Engineers (2004). (in Japanese).

[12] K. Aoki, H. Ueda, Y. Kida, T. Shimoyama and Y. Sonoda: "A trial of GNFS implementation (part I) — summary", 2004 Symposium on Cryptography and Information Security, No. 2B1-3 in SCIS'04, Hotel Sendai Plaza, Sendai, Japan, Technical Group on Information Security (IEICE), pp. 269–273 (2004). (in Japanese).

[13] S. Cavallar, B. Dodson, A. K. Lenstra, W. Lioen, P. L. Montgomery, B. Murphy, H. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam and P. Zimmermann: "Factorization of a 512-bit RSA modulus", Advances in Cryptology — EUROCRYPT 2000 (Ed. by B. Preneel), Vol. 1807 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, New York, pp. 1–18 (2000).

[14] S. Contini: "Factor world!", (http://www.crypto-world.com/FactorWorld.html) (2002).