アルゴリズム 57－2
（1997. 5. 16）

# ブロック分割を用いた素因数分解アルゴリズム

小林 弘忠
東京大学理学部情報科学科

**Abstract**

今日の暗号システムの安全性は，素因数分解問題の難解さの仮定の下に保証されている．現在知られている効率のよい素因数分解アルゴリズムには，複数多項式二次ふるい法・数体ふるい法などがあるが，これらの方法は，ふるいにかける際のメモリ容量制限から，キャッシュミスなどによる実行速度の低下が大きい．そこで，Wambach と Wettig は二段化したブロック分割アルゴリズムを用い，ふるいデータのキャッシュミスを減らし，複数多項式二次ふるい法の実行速度を 2 倍以上にすることに成功した．本論文では，ふるいデータのみならずキャッシュ上にある素数の効率的利用を図り，より速い二段ブロック分割アルゴリズムを構成し，さらに二次キャッシュまでの効率的利用を図った三段ブロック分割アルゴリズムにより，100 桁以上の合成数に対し Wambach と Wettig のアルゴリズムからさらに 30 ～ 40 % の速度向上を得た．

# Factoring Integers with Block Sieving Algorithm

Hirotada Kobayashi
Department of Information Science, University of Tokyo

**Abstract**

The security of present cryptosystems is based on the supposed intractability of the factoring integer problem. Multiple Polynomial Quadratic Sieve (MPQS) and Number Field Sieve (NFS) are the most efficient known algorithms for factoring integers today. Since these algorithms are memory-bounded processes in sieving phase, slowdown caused by cache misses is not ignorable. Wambach and Wettig introduced a double block sieving algorithm to sieving phase of MPQS in order to avoid cache misses in sieve array, and achieved a speedup greater than two. This paper focuses on effective use of not only sieve array but primes in cache to achieve further speedup. Moreover, to make efficient use of both first level cache and second level cache, we introduce triple block sieving algorithm, which is 30-40 % faster than Wambach and Wettig's for 100 digits composite and above.

## 1 導入

素因数分解アルゴリズムとしては，まず，もっとも簡単なものとして，小さな素数から順に割っていく試行割算法 (Trial Division) があげられるが，これは当然ながら効率が悪く，$10^{12}$ より大きい素因数が含まれていると実用的でなくなる．

そこで，現在知られている効率の良い方法であるはいずれも，適当な $z$ に対し $\gcd(N, z)$ を求めて $N$ の素因数を見つけることをもとにしている．アルゴリズムは素因数依存なものと合成数依存なものに大きく二分され，前者の例としては楕円曲線法 [7]，後者の例としては複数多項式二次ふるい法 [10] や数体ふるい法 [6] があげられる．

以下では，複数多項式二次ふるい法を例に，ブロック分割を用いてキャッシュミスなどによる速度低下を防ぐことを考える．

## 2 複数多項式二次ふるい法

### 2.1 基本アルゴリズム

複数多項式二次ふるい法は二次ふるい法 (Pomerance [9]) の改良であるので，まず二次ふるい法について述べる．

奇数合成数 $N$ に対し，$s^2 \equiv t^2 \bmod N$, $s \not\equiv \pm t \bmod N$ なる $s, t\,(s > t)$ を求めれば，$\gcd(N, s+t)$, $\gcd(N, s-t)$ によって $N$ の因数をみつけることができる．このような $s, t$ を効率よくみつけるために，

$$Q(x) = Ax^2 + Bx + C \equiv \{H(x)\}^2 \pmod N$$

なる二次多項式 $Q(x)$ を導入し，$Q(u)$ の完全な素因数分解が得られるような $u$ を複数求めて，そのうちのいくつかを掛け合わせて各素因数に対し指数が偶数になるようにするのが二次ふるい法の概念である．

アルゴリズムは基本的に，

1. $FB = \{p_i \,|\, (N/p_i) = 1,\ p_i$ : 素数, $i = 1, \cdots, R\}$ なる $p_R$ 以下の素数の部分集合 (大きさ $R$) を作る．
2. 二次多項式 $Q(x) = Ax^2 + Bx + C \equiv \{H(x)\}^2 \pmod N$ を定める (係数の決め方は [4] [8] [10] 参照)．
3. ふるいを用いて，$u \in [-M, M]$ で，$FB$ 上で $Q(u)$ の完全な素因数分解が得られるものを集める．
4. 3. で集めた $u$ の部分集合 $U$ で，$\prod_{u \in U} Q(u)$ が平方数になるような $U$ を求める．
5. 4. の $U$ に対しては，

$$s^2 = \prod_{u \in U} Q(u) \equiv_N \prod_{u \in U} \{H(u)\}^2 = t^2$$

となる．

条件を満たす複数の $u$ をみつけるのに，$Q(x)$ として 1 つの多項式のみを用いると，ふるいにかける区間を大きくとらなければならない．そこで，Silverman [10] は複数の多項式を用いることにより，小さい区間で効率良く $u$ をみつけるようにした．これが複数多項式二次ふるい法である．

### 2.2 ふるいの利用

前節のアルゴリズムのステップ 3. で $Q(u)$ を $FB$ 上で素因数分解するのに，全ての $u \in [-M, M]$ に対し，Trial Division を行うのでは効率が悪い．そこで二次ふるい法では，ふるいを利用して，完全な素因数分解が得られそうな $u$ の候補を探し，それらに対してのみ Trial Division を行うことで効率化を図っている．

具体的には，素数 $p \in FB$ に対し，

- $p | Q(u) \quad \Leftrightarrow \quad Q(u) \equiv 0 \bmod p$
  $\Leftrightarrow \quad Q(u \pm kp) \equiv 0 \bmod p \quad k = 0, 1, 2, \ldots$
- $Q(u) \quad = \quad p_1 p_2 \cdots \Leftrightarrow \log Q(u) = \log p_1 + \log p_2 + \ldots$

であるから，

1. $x \in [-M, M]$ に対し，ふるい $S[x] = 0$ と初期化．
2. for all $p_r \in FB$ do {
   $Q(u) \equiv 0 \bmod p_r$ の解 $x_{r,1}, x_{r,2}$ を求める
   for $k = 0, 1, 2, \ldots$ do {
   $S[x_{r,1} \pm kp_r]$, $S[x_{r,2} \pm kp_r]$ に $\log p_r$ を足す
   }
   }

3. $S[u] = \log Q(u) \Leftrightarrow Q(u)$ は $FB$ 上で完全に素因数分解される.

として候補を見つける. $M$ が充分に大きいと, 内側の for 文の繰り返しにおいて, $\lceil s_c/p_r \rceil$ 回アクセスするごとにキャッシュミスが, $\lceil (n_e s_p)/p_r \rceil$ 回アクセスするごとに TLB ミスが起きることになる ($s_c$ : cache size, $s_p$ : page size, $n_e$ : TLB のエントリ数).

# 3 既存のブロック分割アルゴリズム

二次ふるい法の実行時間の 7 割以上はふるいにかける時間であり, 100 桁以上の合成数の素因数を見つける場合, $FB$ の大きさは 50,000 ~ 300,000 以上, $M$ は 10,000,000 ~ 50,000,000 以上になるので, キャッシュミスや TLB ミスによる実行時間低下が無視できない. そこで, Wambach と Wettig [11] はブロック分割アルゴリズムによりアクセスミスを減らすことを考えた.

## 3.1 一段ブロックアルゴリズム

区間 $[0, M[$ を $[0, B[, [B, 2B[, ..., [(k-1)B, M[$ の $k$ 個の区間に分割してふるいにかけることを考える ($k = \lceil M/B \rceil$). 簡単のため, $B|M$ とする. $B \leq n_e s_p$ ととる.

一段ブロックアルゴリズムは図 1 のようになる. 二次方程式の解を繰り返し計算しなくても良いように, 1 ブロック評価し終わるごとに解を更新している. $B \leq n_e s_p$ となるようにとってやれば, TLB ミスは起きない.

## 3.2 二段ブロックアルゴリズム

さらに, 一次キャッシュをより効率的に用いるため, ブロック分割を二段化する.

まず, 区間 $[0, M[$ を $[0, B_1[, [B_1, 2B_1[, ..., [(k_1-1)B_1, M[$ の $k_1$ 個の区間に分割し ($k_1 = \lceil M/B_1 \rceil$), さらにそれぞれをサイズ $B_2$ の小ブロック $k_2$ 個 ($k_2 = \lceil B_1/B_2 \rceil$) に分割する. 簡単のため, $B_1|M$, $B_2|B_1$ とする. $B_2 \leq s_{c_1}$, $B_1 \leq n_e s_p$ ととる.

$FB$ の中の素数を, $p_0, ..., p_{r_m-1}, p_{r_m}, ..., p_{R-1}$ に 2 分割し (実際には比較演算を減らすため 4 分割するのだが, 事実上 2 分割), $p_{r_m}$ は実験的に決める (例えば, $p_{r_m-1} \leq B_1/4 < p_{r_m}$ ).

アルゴリズムは図 2 のようになる.

例えば, 一次キャッシュサイズ 16 kb, $R = 200{,}000$, $M = 2^{25} \simeq 32{,}000{,}000$ とすると, 実験的には $B_1 = 16{,}384$, $r_m \simeq 7{,}000$ ととるのが良く, この場合ブロック $B_1$ 内で

$$\frac{\sum_{r=1}^{7{,}000} M/p_r}{\sum_{r=1}^{200{,}000} M/p_r} \simeq \frac{\int_1^{20 \cdot 7{,}000} M/p \, dp}{\int_1^{20 \cdot 200{,}000} M/p \, dp} \simeq 0.779$$

より, 約 80 % のふるいへのアクセスを完了していることになる (実験的に $p_r \simeq 20r$ であることを用いた).

# 4 ブロック分割アルゴリズムの改良

## 4.1 改良二段ブロックアルゴリズム

Wambach らのアルゴリズムは, ふるいにかける区間に対してのキャッシュの効率的利用に重点を置いているのに対し, キャッシュ内にある素数をも効率的に用いることを目的に二段ブロックアルゴリズムを改良する.

```
for (b = 0; b < k; b++) {
    initialize(S, B);
    for all p_r ∈ FB do {
        d_1 = x_{r,2} − x_{r,1}; d_2 = p_r − d_1; B_p = B − d_1;
        for (x = x_{r,1}; x < B_p; ) {
            S[x] = S[x] + log p_r;
            x = x + d_1;
            S[x] = S[x] + log p_r;
            x = x + d_2;
        }
        if (x < B) {
            S[x] = S[x] + log p_r;
            x = x + d_1;
            x_{r,1} = x − B;
            x_{r,2} = x_{r,1} + d_2;
        } else {
            x_{r,1} = x − B;
            x_{r,2} = x_{r,1} + d_1;
        }
    }
    evaluate(S, B);
}
```

図 1: Single block algorithm

```
for (b_1 = 0; b_1 < k_1; b_1++) {
    initialize(S, B_1);
    for (b_2 = 0; b_2 < k_2; b_2++) {
        sieve [b_2B_2, (b_2 + 1)B_2[ using the single block algorithm
        with primes upto indices r_m (without initialization and evaluation)
    }
    sieve [b_1B_1, (b_1 + 1)B_1[ using the single block algorithm
    with primes with indices from r_m to R (without initialization and evaluation)
    evaluate(S, B_1);
}
```

図 2: Double block algorithm - Wambach & Wettig's version

先と同様, 区間 $[0, M[$ を $[0, B_1[, [B_1, 2B_1[, ..., [(k_1-1)B_1, M[$ の $k_1$ 個の区間に分割し ($k_1 = \lceil M/B_1 \rceil$), さらにそれぞれをサイズ $B_2$ の小ブロック $k_2$ 個 ($k_2 = \lceil B_1/B_2 \rceil$) に分割する. 簡単のため, $B_1|M$, $B_2|B_1$ とする. $B_2 \leq s_{c_1}$, $B_1 \leq n_e s_p$ ととってやればよい.

$FB$ の中の素数を,$p_0, ..., p_{r_s-1}, p_{r_s}, ..., p_{r_l-1}, p_{r_l}, ..., p_{R-1}$ に 3 分割し,$p_{r_s-1} \leq B_2 < p_{r_s}$, $p_{r_l-1} \leq B_1 < p_{r_l}$ とする.

アルゴリズムは図 3 のようになる.

```
for (b1 = 0; b1 < k1; b1++) {
    initialize(S, B1);
    for (b2 = 0; b2 < k2; b2++) {
        sieve [b2B2, (b2 + 1)B2[ with primes upto indices rs
    }
    sieve [b1B1, (b1 + 1)B1[ with primes with indices from rs to rl
}
for (b1 = 0; b1 < k1; b1++) {
    sieve [b1B1, (b1 + 1)B1[ with primes with indices from rl to R
    evaluate(S, B1);
}
```

図 3: New double block algorithm

先と同様, 一次キャッシュサイズ 16 kb, $R = 200,000$, $M = 2^{25} \simeq 32,000,000$ の時を考えると, 実験的には $B_1 = 8,192$, $r_s \simeq 500$, $r_l \simeq 20,000$ ととるのが良く, この場合最初の for ループのブロック $B_1$ 内で

$$\frac{\sum_{r=1}^{500} M/p_r}{\sum_{r=1}^{200,000} M/p_r} \simeq \frac{\int_1^{20\cdot 500} M/p\, dp}{\int_1^{20\cdot 200,000} M/p\, dp} \simeq 0.605$$

より, 約 60 % だけふるいへのアクセスを完了していることになる (実験的に $p_r \simeq 20r$ であることを用いた).

## 4.2 三段ブロックアルゴリズム

さらに, 二次キャッシュの効率的利用も考え, ブロック分割を三段化する.

区間 $[0, M[$ を $[0, B_1[, [B_1, 2B_1[, ..., [(k_1-1)B_1, M[$ の $k_1$ 個の区間に分割し ($k_1 = \lceil M/B_1 \rceil$), さらにそれぞれをサイズ $B_2$ の中ブロック $k_2$ 個 ($k_2 = \lceil B_1/B_2 \rceil$) に分割し, さらにそれぞれをサイズ $B_3$ の小ブロック $k_3$ 個 ($k_3 = \lceil B_2/B_3 \rceil$) に分割する. 簡単のため, $B_1|M$, $B_2|B_1$, $B_3|B_2$ とする. $B_3 \leq s_{c_1}$, $B_2 \leq s_{c_2}$, $B_1 \leq n_e s_p$ ととってやればよい.

この場合, 先の二段ブロックアルゴリズムの $B_1$ はそのまま三段ブロックアルゴリズムの $B_1$ に対応しているが, 先の二段ブロックアルゴリズムの $B_2$ が三段ブロックアルゴリズムの $B_3$ に対応し, 新たに二次キャッシュの効率的利用のために中間に $B_2$ のブロックが出来ることになる.

$FB$ の中の素数を, $p_0, ..., p_{r_s-1}, p_{r_s}, ..., p_{r_m-1}, p_{r_m}, ..., p_{r_l-1}, p_{r_l}, ..., p_{R-1}$ に 4 分割し, $p_{r_s-1} \leq B_3 < p_{r_s}$, $p_{r_m-1} \leq B_2 < p_{r_m}$, $p_{r_l-1} \leq B_1 < p_{r_l}$ とする.

アルゴリズムは図 4 のようになる.

```
for (b1 = 0; b1 < k1; b1++) {
    initialize(S, B1);
    for (b2 = 0; b2 < k2; b2++) {
        for (b3 = 0; b3 < k3; b3++) {
            sieve [b3B3, (b3 + 1)B3[ with primes upto indices rs
        }
        sieve [b2B2, (b2 + 1)B2[ with primes with indices from rs to rm
    }
    sieve [b1B1, (b1 + 1)B1[ with primes with indices from rm to rl
}
for (b1 = 0; b1 < k1; b1++) {
    sieve [b1B1, (b1 + 1)B1[ with primes with indices from rl to R
    evaluate(S, B1);
}
```

図 4: Triple block algorithm

## 5 結果と考察

実験には，100 桁の合成数 RSA100p8681 及び $7^{194}+1$ (Cunningham project [3]) の因数である 116 桁の合成数を用い，それぞれ 40,000 個，200,000 個の素数要素からなる $FB$ と，$M = 2^{20}$，$M = 32 \cdot 2^{20}$ の区間でふるいにかけ，1 多項式に対し要する平均時間を測定した (表 1).

実験に用いたマシンは Ultra Sparc 166 Mhz, Memory 256 MB, 一次データキャッシュ 16 KB, 二次キャッシュ 512 KB で，TLB はページサイズ 8 KB × 64 エントリ であるので，一段ブロックアルゴリズムの $B = 512$ KB，二段ブロックアルゴリズムの $B_1 = 512$ KB, $B_2 = 16$ KB，改良二段ブロックアルゴリズムの $B_1 = 512$ KB, $B_2 = 8$ KB，三段ブロックアルゴリズム の $B_1 = 512$ KB, $B_2 = 256$ KB, $B_3 = 8$ KB とした.

| *algorithm* | **RSA100p8681** | | | ***C* 116** | | |
|---|---|---|---|---|---|---|
| | *sieve time* (sec) | *sieve speedup* | *total speedup* | *sieve time* (sec) | *sieve speedup* | *total speedup* |
| naive | 10.993906 | 1.00 | 1.00 | 38.304821 | 1.00 | 1.00 |
| single block | 4.209109 | 2.61 | 2.49 | 18.178340 | 2.11 | 2.06 |
| double block | 3.347519 | 3.28 | 3.07 | 14.381387 | 2.66 | 2.57 |
| our double block | 2.615346 | 4.20 | 3.83 | 12.296861 | 3.12 | 2.98 |
| our triple block | 2.360920 | 4.66 | 4.19 | 11.338017 | 3.38 | 3.21 |

表 1: speedup by block sieving

1 多項式に対し，sieving phase 以外に，$Q(x)$ の値を計算したり，解を求めたり，$FB$ 上での trial division

をしたりするのに，RSA100p8681 で平均 0.34 秒，$C$ 116 で平均 0.85 秒かかる．これを考慮しても，triple block sieving は naive sieving に比べ RSA100p8681 で 4 倍以上，$C$ 116 で 3 倍以上の速度向上がみられ，Wambach らの double block sieving に比べても，約 30 ～ 40 % の速度向上がみられる．

なお，これらの測定時間はあくまで 1 多項式にかかる時間であり，実際に最後まで素因数分解しようとすると，single processor では 1 ヵ月以上は少なくともかかる計算になるので，時間の制約上最後まで素因数分解することは出来なかった．

実際に素因数分解できた最大サイズの合成数は，$74^{88}+1$ (Brent, Montgomery and te Riele's extensions to the Cunningham Project [2]) の因数である 87 桁の合成数で，1996 年 11 月現在未分解のもので，

108 0239 0303 3928 2189 8961 4308 0011 0699 4662 9938
7577 0504 8320 8771 8439 1038 1876 9755 5669 9587 0337

⇓

929 9722 9516 9143 2891 4935 6524 9009 *
1161 5819 4814 0517 8834 6139 7748 2815 2836 0191 6151 0165 2391 2593

となった．

これらのブロック分割アルゴリズムの効果を $FB$ の大きさを変えて調べてみた．$R = 10{,}000$～$250{,}000$ に対する各アルゴリズムの速度向上は 図 5 のようになり，三段ブロックアルゴリズムは Wambach らの二段ブロックアルゴリズムより常に 30 ～ 40 % 速いことが分かるが，一方で各アルゴリズムとも $FB$ のサイズが大きくなるとブロック分割の効果が薄れてくることが分かる．これは，$FB$ が大きくなると大きな歩幅でふるいに対しアクセスする割合が高くなり，アクセスミスが起き易くなるためと考えられる．これに対する対策は 2,3 考えたが，いずれも本質的な解決にはなっていない．

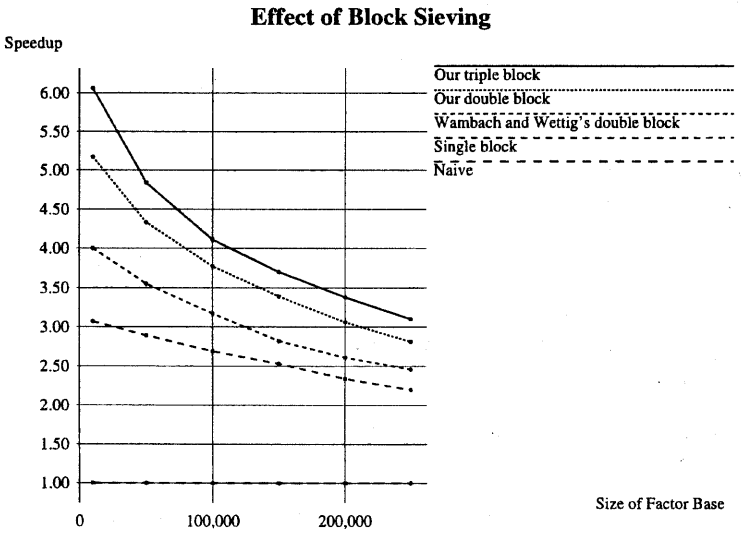


図 5: Relation between the effect of block sieving and the factor base size

## 6 まとめと今後の課題

本論文では，素因数分解問題に対する複数多項式二次ふるい法において，ふるいにかける際のブロック分割アルゴリズムを改良し，よりキャッシュミスの起きにくい，速いアルゴリズムを提案した．具体的には，ふるいだけでなく素数に対するアクセスミスも減らすことを念頭に，二次キャッシュまでの効率的利用を図って三段ブロック分割アルゴリズムを導入した．

しかしながらこのアルゴリズムにもまだ，前節で見たように $FB$ が大きくなるとブロック分割をした効果が薄れるという欠点があるので，これを解決することが課題の一つである．

また，構造的にブロック分割アルゴリズムは並列化に向いているので，この実現もまた，課題であろう．

さらに，数体ふるい法への応用も可能なので，これも取り組むべきことの一つである．

## 参考文献


[1] D. Atkins, M. Graff, A. K. Lenstra & P. C. Leyland, "The Magic Words are Squeamish Ossifrage," Lecture Notes in Computer Science, Vol.917, pp.263-277, 1995.

[2] R. P. Brent & H. J. J. te Riele, "Factorizations of $a^n \pm 1$, $13 \leq a < 100$ , " Report NM-R9212, Centrum voor Wiskunde en Informatica, Amsterdam, June 1992.

[3] J. P. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman & S. S. Wagstaff, Jr., *Factorizations of* $b^n \pm 1$ *for* $b = 2, 3, 5, 6, 7, 10, 12$, *Up to High Powers*, Contemp. Math., Vol.22, Amer. Math. Soc., Providence, R. I., 1983.

[4] F. Damm, F. P. Heider & G. Wambach, "Factoring Integers Above 100 Digits Using Hypercube MPQS," Angewandte Mathematik und Informatik Universität zu Köln, Report No. 94.155, Mar. 1994.

[5] 小山謙二 & 静谷啓樹, "素因数分解と離散対数アルゴリズム," 情報処理, Vol.34, No.2, pp.157-169, Feb. 1993.

[6] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse & J. M. Pollard, "The number field sieve," Proc. 22nd STOC, pp.564-572, 1990.

[7] H. W. Lenstra, Jr., "Factoring Integers with Elliptic Curves," Annals of Mathematics, Vol.126, pp.649-673, 1987.

[8] R. Peralta, "A Quadratic Sieve on the $n$-Dimentional Cube," Lecture Notes in Computer Science, Vol.740, pp.324-332, 1993.

[9] C. Pomerance, "The Quadratic Sieve Factoring Algorithm," Lecture Notes in Computer Science, Vol.209, pp.169-182, 1985.

[10] R. D. Silverman, "The Multiple Polynomial Quadratic Sieve," Math. Comp., Vol.48, No.177, pp.329-339, Jan. 1987.

[11] G. Wambach & H. Wettig, "Block Sieving Algorithms," Angewandte Mathematik und Informatik Universität zu Köln, Report No. 95.190, May 1995.