Sieving Using Bucket Sort*

Kazumaro Aoki and Hiroki Ueda

NTT, 1-1 Hikarinooka, Yokosuka-shi, Kanagawa-ken, 239-0847 Japan
{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.

1 Introduction

The integer factoring problem is one of the most important topics for public-key cryptography, because the RSA cryptosystem is the most widely used public-key cryptosystem, and its security is based on the difficulty of the integer factoring problem. Over a few hundred bits, the number field sieve [1] is currently the most fastest algorithm to factor an RSA modulus.

The number field sieve consists of many steps. It is known that the sieving step is theoretically and experimentally the most time-consuming step. It is noted that a straightforward implementation of the sieving step on a PC causes a long delay in memory reading and writing, and the sieving program is several dozen times faster if all memory accesses utilize the cache memory.

This paper focuses on memory access in the software implementation of the sieving step on a PC, and introduces an algorithm that reduces the number of cache hit misses. The experimental results confirm that the proposed sieving algorithm is several times faster than that in the straightforward implementation.

2 Preliminaries

2.1 Number Field Sieve

This section briefly describes the number field sieve algorithm that is relevant to the scope of the paper. Details regarding this algorithm can be found in (e.g. [1]).

* This work is supported in part by a consignment research from the Telecommunications Advancement Organization of Japan (now the National Institute of Information and Communications Technology) and by the CRYPTREC project.

Let N be a composite number and it will be factored. Find an irreducible polynomial $f(X) \in \mathbb{Z}[X]$ and its root M such that $f(M) \equiv 0 \pmod{N}$. The purpose of the sieving step is to collect many coprime pairs $(a, b) \in \mathbb{Z}^2$ such that $N_R(a, b) = |a + bM|$ is B_R-smooth and $N_A(a, b) = |(-b)^{\deg f} f(-a/b)|$ is B_A-smooth[1]. Such a coprime, (a, b), is called a *relation*.

We describe the line-by-line sieve (hereafter we simply referred to as *line sieve*) as Algorithm 1, and it is the most basic algorithm used to find relations. Hereafter, we omit the details on the algebraic side, because very similar algorithms are used for the algebraic side. Algorithm 1 assumes that $2H_a$ elements are allocated for array $\mathtt{S}$. The sieving region may be divided if $2H_a$ is greater than the suitable size for the implementation platform. The size of each element, $\mathtt{S}[a]$, is typically 1 byte, and the base for $\log p$ is selected such that it does not to exceed the maximum representable value of $\mathtt{S}[a]$. In Step 8 in the inset,

Algorithm 1 (line sieve for rational side (basic version)).

1: **for** $b \leftarrow 1$ **to** H_b
2:     for all a $(-H_a \leq a < H_a)$, initialize $\mathtt{S}[a]$ to $\log N_R(a, b)$
3:     **for prime** $p \leftarrow 2$ **to** B_R
4:         Compute $a \geq -H_a$ as the first sieving point depending on b and p
5:         **while** $a < H_a$
6:             $\mathtt{S}[a] \leftarrow \mathtt{S}[a] - \log p$
7:             $a \leftarrow a + p$
8:     Completely factor $N_R(a, b)$ for all a if $\mathtt{S}[a] <$ some threshold

the threshold is determined by considering the error generated by the logarithm rounded to the nearest integer in Steps 2 and 6, and the omission of prime powers[2].

2.2 Large Prime Variation

If B_R is close to or greater than H_a, the while-loop in Step 5 is hardly activated, and the first sieving point computation in Step 4 may dominate the sieving time. For this case, we can use the *large prime variation*. The changes compared to the basic version are as follows:

1. Set the bound for p at Step 3 to B_R^L $(< B_R)$.
2. Relax the threshold at Step 8 in Algorithm 1.

The faster the primality testing and factoring for small integers greater than B_R^L become available, the more relaxed the threshold can become.

[1] "x is y-smooth" means that all prime factors of x are less than or equal to y.
[2] By regarding prime power p^e as prime and $\log p^e$ as $\log p$, prime powers can be easily incorporated into Algorithm 1.

Based on the experience, the most time-consuming part in large prime variation is reading and writing to memory to update $\mathtt{S}[a]$ in Step 6. This paper optimizes the memory read/write process.

2.3 Memory Latency of a PC

Recent PCs have incorporated cache memory, and cache memory can usually be classified into several levels. A low level cache represents fast access but low capacity. For better understanding, we provide an example. Let us consider the Pentium 4 memory characteristics for logical operations performed by general purpose registers as shown in Table 1.

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

	Line size	Size	Latency
Register	(4 B)	32 B	$\frac{1}{2}$ processor cycle
Level 1 cache	64 B	8 KB	2 processor cycles
Level 2 cache	64 B+64 B	512 KB	7 processor cycles
Main memory	(4 KB)	≈1 GB	12 processor cycles + 6-12 bus cycles

The memory system in a PC is constructed to provide good performance for continuous address access, that is, random address access is very poor. A line sieve algorithm updates $\mathtt{S}[a]$ by step p in Step 6 in Algorithm 1. When p is greater than the size of cache memory, the updates seem to be random access. A read from the main memory requires at least $12 + 6 \times (2.53/0.533) = 40.5$ processor cycles, where the Pentium 4 frequency is 2.53 GHz and FSB is 533 MHz, according to Table 1. However, the user probably feels that the time required for main memory access requires more processor cycles. An experiment shows that the time for a random read from the main memory requires several hundred processor cycles.

2.4 Previous Work

Sieving can be considered as waiting for memory because other steps in the innermost loop are small and very simple, according to Steps 5 to 7 in Algorithm 1. To overcome cache hit misses, [3] proposed the *block sieving* algorithm. There are two differences between the basic version of the line sieve in Algorithm 1 and the block sieving algorithm: the addition of Algorithm 2 between Steps 2 and 3, and the initial p in Step 3 is modified to the smallest prime greater than B_R^S. The block sieving algorithm classifies factor base primes into *smallish primes* ($\in (0, B^S]$) and *largish primes* ($\in (B^S, B^L]$), and updates each small region whose size is H_a^S by smallish primes. To achieve better performance, H_a^S and B_R^S are set to approximately the size of the cache memory. Note that the computation of the first sieving point in Step 3 in Algorithm 2 can be omitted if the

Algorithm 2 (Additional steps from line sieve to block sieving algorithm).

1: **for** $a^S \leftarrow -H_a$ **to** H_a **step** $+H_a^S$
2:     **for prime** $p \leftarrow 2$ **to** B_R^S
3:     Compute $a \geq a^S$ as the first sieving point
4:     **while** $a < a^S + H_a^S$
5:         $\mathtt{S}[a] \leftarrow \mathtt{S}[a] - \log p$
6:         $a \leftarrow a + p$

last sieving point computed in Step 4 is available. Focusing on the memory hierarchy, the performance of the sieving step may be better optimized in order to consider more parameters in classifying smallish primes in some environments.

3 Sieving Using Bucket Sort

The number of cache hit misses for smallish primes greatly decreases using the block sieving algorithm described in Sect. 2.4; however, the sieving for largish primes still generate many cache hit misses. This section describes the reduction in the number of cache hit misses for largish primes using the bucket sorting algorithm [4, Sect. 5.2.5].

As mentioned in Sect. 2.3, memory updates between close addresses are not penalized, and the $\log p$ minuses which are memory update operations are commutative. Sorting $(a, \log p)$ using key a can reduce the number of cache hit misses; however, the sorting should be done very quickly, because the number of $\mathtt{S}[a]$ updates is roughly $2H_a(\log\log B^L - \log\log B^S)$, that is, it is almost linear to H_a. While complete sorting is not required and recent PC models have very large memory capacity, we use the bucket sorting algorithm to address this issue.

3.1 Proposed Algorithm

The proposed algorithm replaces the largish prime sieving in Algorithm 1, that is, the algorithm has the same function as Algorithm 1 for sieving largish primes.

The algorithm is based on bucket sorting. Let n be the number of buckets, and r be $\left\lceil \frac{n_{\mathtt{S}}}{n} \right\rceil$, where $n_{\mathtt{S}}$ denotes the number of elements in $\mathtt{S}$. Note that $n_{\mathtt{S}} = 2H_a$ for Algorithm 1. The algorithm comprises the continuous runs of Algorithms 3 and 4.

Algorithm 3 throws $(a, \log p)$ in the buckets, and Algorithm 4 updates $\mathtt{S}[a]$ using the elements in the buckets.

3.2 Why Can Proposed Algorithm Hit Cache Memory?

Figure 1 forms the basis for the following discussion. We first consider Algorithm 4. All elements in a bucket will only updates the memory in range r.

Algorithm 3.

1: Let all buckets empty
2: **for prime** $p \leftarrow B_R^S + 1$ **to** B_R^L
3: Compute $a \geq -H_a$ as the first sieving point
4: **while** $a < H_a$
5: Store $(a, \log p)$ to the $\left\lfloor \dfrac{a + H_a}{r} \right\rfloor$th bucket
6: $a \leftarrow a + p$

Algorithm 4.

1: **for** all buckets that are numbered i $(0 \leq i < n)$
2: **for** all $(a, \log p)$ in the bucket i
3: $\mathtt{S}[a] \leftarrow \mathtt{S}[a] - \log p$

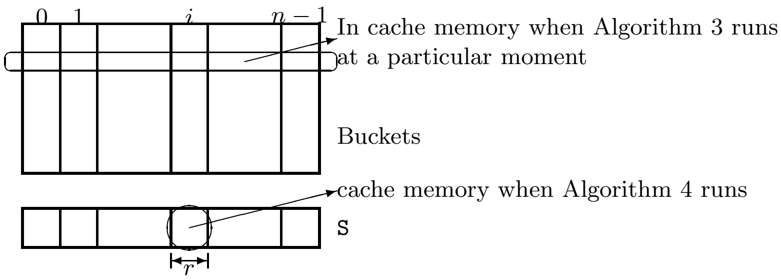

Fig. 1. Memory usage for buckets and S

Thus,

$$r \times (\text{Size of each } \mathtt{S}[a]) \leq (\text{Size of cache memory}) \tag{1}$$

should hold. Next, we consider Algorithm 3. For each bucket, the addresses for memory writes are continuous. It is sufficient if

$$n \times (\text{Size of cache line}) \leq (\text{Size of cache memory}) \tag{2}$$

holds. Note that the cache memory can only be updated in units called cache lines. We assume that the size of $(a, \log p)$ is less than the size of a cache line. When combining (1) and (2), n exists if

$$n_{\mathtt{S}} \times (\text{Size of each } \mathtt{S}[a]) = (\text{Size of } \mathtt{S}[\bullet]) \leq \frac{(\text{Size of cache memory})^2}{(\text{Size of cache line})} \tag{3}$$

holds.

Let us consider a typical parameter using Table 1. The size of the cache memory is 512 KB, and the size of the cache line is 128 B. Therefore, the right hand side of (3) is 2^{31} B. If we allocate each $\mathtt{S}[a]$ as 1 B, then $\mathtt{S}$ can occupy up to 2 GB. This means that the proposed algorithm is effective for most PCs. The proposed algorithm increases the number of memory accesses, but dramatically reduces the number of cache hit misses with appropriate prefetching.

3.3 Related Work

[5], which follows the inspiring work [6], independently proposed sieving hardware, which sorts $(a, \log p)$. The paper does not consider the cache memory; however, their algorithm is similar in that *sieving* is converted to *sorting*.

4 Optimizations and Improvements

This section considers optimization techniques and improvements to the proposed algorithm.

4.1 $(a, \log p)$ Size Reduction

The size of a stored in a bucket can be reduced. $a' = a + H_a \bmod r$ is sufficient to recover a, because $a = ir + a' - H_a$ for the ith bucket.

Moreover, the number of bits for $\log p$ can be reduced to 1 bit, because $(a, \log p)$ can be generated in ascending order on p and $\log p$ in a bucket increases very slow.

4.2 Number of Buckets

For efficient computation of Step 5 in Algorithm 3 and the technique described in Sect. 4.1, r should be a power of 2 on most PCs.

4.3 Hierarchical Buckets

Considering the idea of radix sort and cache hierarchy, Algorithm 4 can be modified to Algorithms 3 and 4 using smaller buckets.

4.4 Reduction in Memory for Buckets

Consider the case that a PC does not have enough memory to allocate buckets to store all $(a, \log p)$s. Whenever a bucket is full at Step 5 in Algorithm 3, call Algorithm 4 and empty the buckets.

4.5 Reduction in Sieving Memory $\mathtt{S}$

First, perform sieving for largish primes using Algorithms 3 and 4. When executing Algorithm 4, smallish prime can be sieved between Steps 1 and 2. In the ith bucket, a is in $[ir - H_a, (i+1)r - H_a)$. Thus, r elements for $\mathtt{S}[a]$ are sufficient for the ith bucket.

Note that this idea cannot be used with the idea described in Sect. 4.4.

4.6 Bucket Reuse for Trial Division

The trial sieving algorithm [7] was proposed to reduce the time in Step 8 in Algorithm 1. The algorithm acts almost the same as the sieving algorithm discussed above, but it only considers a small set of (a, b). When filling buckets in Algorithm 3, store p in addition to $(a, \log p)$, and the buckets can be used for trial sieving. This can reduce the computational cost of the first sieving points for largish primes. However, storing p probably doubles the memory allocation for the buckets. It may be a good idea to avoid storing small primes that are classified as largish primes.

4.7 Application to Lattice Sieve

The idea behind the proposed algorithm can be applied to any algorithm if the memory update operation is commutative. There are no problems in using the proposed algorithm for the lattice sieve.

4.8 Tiny Primes

[8, p.334] suggests that $\mathtt{S}[a]$ is initialized by the logarithm of tiny primes. It can be efficiently achieved by the following idea. First, compute the sieving pattern for tiny primes, which are less than B^T, and their small powers. Once the pattern is computed, the initialization of $\mathtt{S}[a]$ can be done by duplicating the pattern by adjusting the correct starting position.

5 Implementation on Pentium 4

We implemented Algorithms 3 and 4 in the lattice sieve using all the techniques in Sect. 4 except for Sect. 4.4 and last haf of Sect. 4.1 on a Pentium 4 (Northwood) with 1 GB main memory and 533 MHz FSB. The specifications are the same as those described in Table 1. The prime bounds are described in Table 2. These names are from [9]. We tried to obtain the best Bs using the factor base

Table 2. Prime Bounds and Algorithms

Range	Name	Algorithm
$p \le B^T$	p: Tiny prime	Sieving pattern
$B^T < p \le B^S$	p: Smallish prime	Block sieving
$B^S < p \le B^L$	p: Largish prime	Bucket sorting
$B^L < p \le B$	p: Large prime	Primality testing and factoring

parameter for c158 as described in [10].

5.1 Parameter Selection

We assign 1 B for $\log p$ and 4 B for each $(a, \log p)$, because the smallest memory read and write unit is 1 B and the basic memory data unit is 4 B for the Pentium 4.

On the factorization of c158, the sieving rectangle was $2H_c \times H_d = 2^{14} \times 2^{13}$. To translate the rectangle to a line sieve case, we can interpret $2H_a = 2^{14} \times 2^{13} = 2^{27}$. The large primes in each relation and the values of B_R^L and B_A^L are unclear. Therefore, we select two large primes for both sides in each relation, and set $B_R^L = 30 \times 10^6$, $B_A^L = 0.9 \times Q$, and $B_R^S = B_A^S = 512 \times 2^{10}$, where Q denotes the special-Q according to our factoring code, the primality testing for large prime products, factor base bound for the line sieve, and the size of level 2 cache. We tried the depths of 1, 2, and 3 for the hierarchical buckets with all powers of 2 for r, and found that the best hierarchy depth is 2. Surprisingly, the best rs are not the combination of the size of the level 2 cache and level 1 cache, but 2 MB and 256 KB.

Next, we tried to find the best B_R^S and B_A^S. Based on dozens of experments, we find that $B_R^S = 2H_c$ and $B_A^S = 5H_c$ achieve almost the best performance.

Remark 1 We sieve prime powers less than $\sqrt{B^L}$, and select $B_R^T = B_A^T = 5$.

Remark 2 We classify smallish primes into small sets taking into account the size of the caches and sieving range.

Remark 3 After executing Algorithm 3, the numbers of elements in each bucket are roughly the same. We found that a 2% difference is the largest in our experiments.

Remark 4 We used base-2 Solovay-Strassen primality testing [11, pp.90–91], and ρ [11, pp.177–183] and SQUFOF [11, pp.186–193] as the factoring algorithm for large primes.

5.2 Factoring Example

We factor 164-digit cofactor c164 in $2^{1826}+1$ using GNFS, and 248-digit cofactor c248 in $2^{1642}+1$ using SNFS employing the above implementation. Refer to the Appendix for detailed information. The parameters used in the factoring of c164 and c248 are summarized in Table 3. For comparison purposes, Table 3 also includes the parameters used in the factoring of RSA-512 [12].

Table 3. Factoring Parameters for Lattice Sieve

	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
c248	16 K	8 K	0.95Q	100 m	4 g	200 m	10.2 m	2+2	22 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: Generated relations per MIPS year

The proposed siever yields more relations per MIPS year despite that c164 is larger than RSA-512. However, a straightforward comparison should be avoided because the characteristics of computers used for the above factoring are quite different, and MIPS is not optimal for comparing the sieving complexity.

Remark 1 The lattice siever used for RSA-512 is intended to factor RSA-130 [12, Sect. 3.2].

Remark 2 We timed MIPS using the output of a "BYTE benchmark." We obtained 3969679.6 lps for Dhrystone 2 without register variables. Thus, MIPS is computed by $3969679.6/1767 \approx 2246.6$. This number is used for c164 and c248 in column rel/MY.

Remark 3 We noticed that numbers larger than RSA-512 such as RSA-576 are already factored using GNFS [13] and that their siever seems faster than one that was used for RSA-512. However, not enough information is provided to estimate the timings. We used the records that were published and the largest values [12].

6 Conclusion

We proposed a sieving algorithm that cleverly uses the cache memory. The algorithm accelerates the memory update processes in the sieving step to several times faster than that of the simple $\log p$ subtraction. Moreover, we implemented the proposed algorithm in the lattice sieve on a Pentium 4, and successfully factored a 164-digit number using GNFS, and a 248-digit number using SNFS.

Acknowledgments

The authors gratefully thank Prof. Yuji Kida for fruitful discussions, his ideas regarding Sect. 4.5 and the last half of Sect. 4.6. Moreover, the authors thank him for his contribution in approximating B^S in Sect. 5.1. We also thank Dr. Takeshi Shimoyama of Fujitsu Labs and Dr. Soichi Furuya of Hitachi for suggesting a hint for reducing the number of bits for $\log p$ in Sect. 4.1.

References

1. Lenstra, A.K., Lenstra, Jr., H.W., eds.: The development of the number field sieve. Volume 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. Wambach, G., Wettig, H.: Block sieving algorithms. Technical Report 190, Informatik, Universität zu Köln (1995) (`http://www.zaik.uni-koeln.de/~paper/index.html?show=zpr95-190`).
4. Knuth, D.E.: Sorting and Searching. Second edn. Volume 3 of The Art of Computer Programming. Addison Wesley (1998)
5. Geiselmann, W., Steinwandt, R.: A dedicated sieving hardware. In Desmedt, Y.G., ed.: Public Key Cryptography — PKC2003. Volume 2567 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York (2003) 254–266
6. Bernstein, D.J.: Circuits for integer factorizaion: a proposal. (available at `http://cr.yp.to/factorization.html#nfscircuit`) (2002)
7. Golliver, R.A., Lenstra, A.K., McCurley, K.S.: Lattice sieving and trial division. In Adleman, L.M., Huang, M.D., eds.: Algorithmic Number Theory — First International Symposium, ANTS-I. Volume 877 of Lecture Notes in Computer Science., Berlin, Heidelberg, New York, Springer-Verlag (1994) 18–27
8. Silverman, R.D.: The multiple polynomial quadratic sieve. Mathematics of Computation **48** (1987) 329–339
9. Shamir, A., Tromer, E.: Factoring large numbers with the TWIRL device. In Boneh, D., ed.: Advances in Cryptology — CRYPTO 2003. Volume 2729 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York (2003) 1–26
10. Bahr, F., Franke, J., Kleinjung, T.: Factorization of 158-digit cofactor of $2^{953}+1$. (available at `http://www.crypto-world.com/announcements/c158.txt`) (2002)
11. Riesel, H.: Prime Numbers and Computer Methods for Factorization. Second edn. Volume 126 of Progress in Mathematics. Birkhäuser, Boston, Basel, Berlin (1993)
12. Cavallar, S., Dodson, B., Lenstra, A.K., Lioen, W., Montgomery, P.L., Murphy, B., te Riele, H., Aardal, K., Gilchrist, J., Guillerm, G., Leyland, P., Marchand, J., Morain, F., Muffett, A., Putnam, C., Zimmermann, P.: Factorization of a 512-bit RSA modulus. In Preneel, B., ed.: Advances in Cryptology — EUROCRYPT 2000. Volume 1807 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York (2000) 1–18
13. Contini, S.: Factor world! (`http://www.crypto-world.com/FactorWorld.html`) (2002)

14. Wagstaff, S.S.: The Cunningham Project. (2003) (`http://www.cerias.purdue.edu/homes/ssw/cun/`).
15. Aoki, K., Kida, Y., Shimoyama, T., Sonoda, Y., Ueda, H.: GNFS164. (`http://www.rkmath.rikkyo.ac.jp/~kida/gnfs164e.htm`) (2003)
16. Aoki, K., Kida, Y., Shimoyama, T., Sonoda, Y., Ueda, H.: SNFS248. (`http://www.rkmath.rikkyo.ac.jp/~kida/snfs248e.htm`) (2004)

Appendix: Factoring parameters and statistics for c164 and c248

c164 and c248 are selected from Cunningham table [14].

c164 is the 164-digit cofactor of $2^{1826}+1$. $2^{1826}+1$ can be trivially factored to $2,1826\mathrm{L} \times 2,1826\mathrm{M}$, where $2,1826\mathrm{L} = 2^{913} - 2^{457} + 1$, and its several factors are already known:

$$\begin{aligned}
2,1826\mathrm{L} = {} & 997 \times 2113 \times 10957 \times 46202197673 \times 209957719973 \\
& \times 457905185813813 \times 9118425814963735020084050069 \\
& \times 758984045239765414366290480154514089 \times \mathrm{c164}
\end{aligned}$$

c164 is factored into two primes, $\mathrm{p68} \times \mathrm{p97}$, where

$$\begin{aligned}
\mathrm{p68} = {} & 3433464488618244465 \\
& 4627300892424208463432708978955977121586409225484 9.
\end{aligned}$$

c248 is the 248-digit cofactor of $2^{1642}+1$. $2^{1642}+1$ can be trivially factored to $2,1642\mathrm{L} \times 2,1642\mathrm{M}$, where $2,1642\mathrm{M} = \mathrm{c248} = 2^{821} + 2^{411} + 1$. c248 is factored into two primes, $\mathrm{p88} \times \mathrm{p160}$, where

$$\begin{aligned}
\mathrm{p88} = {} & 7505293746011641766492467854893261603 6 \\
& 640381023147128390479077762437121481797484501905 33.
\end{aligned}$$

We used the polynomials described in Fig. 2 to factor c164 and c248.

$$\begin{aligned}
\mathrm{c164poly} = {} & \\
& 8293702863045600\ x^5 \\
& + 5627796025215486707\ x^4 \\
& + 5575245564273099319021 11\ x^3 \\
& + 176917216602508818430161036\ x^2 \\
& - 1360117320289954843293521913194 9\ x \\
& - 1217162229047624149744498001231102 1 \\
M = {} & 4112687759327257525969391848 46
\end{aligned}
\qquad
\begin{aligned}
\mathrm{c248poly} = {} & \\
& x^6 \\
& + 2\ x^3 \\
& + 2 \\
M = {} & 2^{137}
\end{aligned}$$

Fig. 2. Polynomials used to factor c164 and c248

Statistics are summarized in Table 4. CPU years for sieving are converted for the Pentium 4 2.53 GHz. Line sieve is used for c164 factoring, and it yields 49 m relations. Free relations are not used for both factorings. Linear algebra is computed by a 16 PC cluster with GbE using block Lanczos with 128-bit block. The Pentium 4 is used for both factoring, but its frequency is about 2.53 GHz for c164 and 3.2 GHz for c248. The programs used for the factoring are basically the same except that minor improvements are included for c248. More detailed information can be found at [15, 16].

Table 4. Statistics

	Sieve		Linear algebra		
	CPU years	Yields	Matrix size	Row weight	Calendar days
c164	7	458 m	7.5 m	167	12
c248	8.2	558 m	7.4 m	208	9.5