ANGEWANDTE MATHEMATIK UND INFORMATIK
UNIVERSITÄT ZU KÖLN

Report No. 95.190

Block Sieving Algorithms

by

Georg Wambach and Hannes Wettig

May 1995

Institut für Informatik
Universität zu Köln
Pohligstraße 1
D-50969 Köln
e-mail: gw @ informatik.uni-koeln.de

Block Sieving Algorithms

Georg Wambach and Hannes Wettig
University of Cologne

May 3, 1995

Abstract

Quite similiar to the Sieve of Erastosthenes, the best-known general algorithms for factoring large numbers today are memory-bounded processes. We develop three variations of the sieving phase and discuss them in detail. The fastest modification is tailored to RISC processors and therefore especially suited for modern workstations and massively parallel supercomputers. For a 116 decimal digit composite number we achieved a speedup greater than two on an IBM RS/6000 250 workstation.

Introduction

Today, in High Performance Computing memory access is one of the biggest problems. Both memory latency and memory bandwidth have not achieved the same growth as processor power during the last decade. Especially the use of fast RISC processors in low-cost workstations with inexpensive RAM, and in massively parallel supercomputers (e.g. Parsytec GC/PowerPlus, IBM SP-series, Cray T3D) usually lack a second-level cache and expensive static RAM with low access time. Everybody who has implemented a matrix-vector multiplication knows about the memory access problems and how to overcome them by blockoriented operations and probably interchanging loops, for example. It is not uncommon to expect a factor of five when optimizing memory access patterns.

The Multiple Polynomial Quadratic Sieve (QS) and the General Number Field Sieve (NFS) are the best known algorithms for factoring large numbers today. Both are general algorithms, i.e. their running time depends on the size of the number to be factored and not on the size of the smallest factor (unlike trial division, e.g.). Factoring large numbers is still the only way to estimate the security of the RSA public-key cryptosystem. Here, a public RSA-modulus N is the product $N = p \cdot q$ of two large primes, where p and q are kept secret. If one is able to find p or q given N, the RSA-modulus N is broken. There are no other methods known to attack RSA-cryptosystems today. Using smaller RSA-moduli in praxis means less space needed for key-management, and less time needed for encryption and decryption, but also less security. Typically one chooses the length of the RSA-modulus such that the estimated time needed for factoring it (with a reasonable environment) exceeds the RSA-modulus' lifetime, which may be one week, six months, a year or more (this is usually the case for hardware implementations). Hence, it is important to know how fast (and how expensive) RSA-moduli can be factored today.

Even a running time improvement of, say 30%, is important: It makes a great difference if you will need seven months or five months to factor a RSA-modulus which is six months valid. Concerning resource consumption, it may save you millions of dollars. Remember the rumor the fall of RSA-129 in 1994 creates? Funny enough, the work estimates for this number [1] have been smallest (1.700 MIPS-years) for a pretty outdated machine (SUN 3/50).

The hot-spot and bottleneck of both QS and NFS is the sieving phase. The sieving phase is similiar to the archaic Sieve of Erastosthenes, where one jumps through a sieve using steps of (different) primes. The difficulty arises because memory is accessed in *irregular* patterns, i.e. every p-th location, p a prime, for all primes less than a certain bound.

In this paper we present a new double block algorithm for sieving. While this algorithm does not improve the number of operations in the sieving phase, it has great influence on the running time. Of course, the running time improvement is machine dependent. However, because the number of RISC processors (or more general with low clock cycles per instruction) will increase in the future, and this is the kind of machine where the highest improvement is possible, we think it is worthwhile to report our experiences. It is surprising that despite considerable bookkeeping operations, the gain resulting from memory accesses still predominates. With this technique we were able to speed up the sieving time for for a 116 decimal digit composite number using QS on a PowerPC-601 machine by a factor of 2,3. 116 decimal digits may be the limit where NFS starts beating QS, but our variations apply to NFS as well. Additionally, we expect to see even bigger improvements on the new multi-chip modules, which have a large second level cache included.

Section 1 contains a short description of QS neccessary for the following, and the notations we used for hardware machinery. In section 2 we will develop the double block algorithm and estimate its running time. Section 3 gives an example, namely, sieving one polynomial for an 116 decimal digit composite. In section 4 we discuss other improvements which we have tested, and make more remarks concering the implementation and the application to NFS.

1 Preliminaries and Notation

First we will sketch the Multiple Polynomial Quadratic Sieve (QS) algorithm in a simplified version with emphasis on the sieving phase. For a complete description see [7, 8, 10].

Let N be the composite integer to be factored. After choosing a factorbase FB of R primes p_r, $0 \le r < R$, and the sievelength M, a lot of quadratic polynomials $Q(X) = a^2X^2 + 2bX + c$ with $b^2 - N = a^2c$ are generated. It follows that $Q(X) \equiv (a^2X + b)a^{-2} \bmod N$. For every such polynomial the roots $x_{r,1}, x_{r,2}$ of $Q(X) \bmod p_r$ must be computed. Then the interval $[-M, M[\cap\mathbb{Z}$ will be sieved (see below). The goal of the sieving phase is to find many locations x where $Q(x)$ factorizes completely over FB:

$$Q(x) = (-1)^s \prod_{r=0}^{R-1} p_r^{e_r} \equiv z_x^2 \bmod N$$

Elementary number theory tells us that $p_r | Q(x)$ if and only if $Q(x) \equiv 0 \bmod p_r$ which is equivalent to $x \equiv x_{r,1}$ or $x_{r,2} \bmod p_r$. Instead of computing $Q(x)$ for every $x \in [-M, M[\cap\mathbb{Z}$ and trial

dividing its value over the primes in FB, QS uses a kind of reversal approach. Let S be an array of length M representing $[0, M[\cap\mathbb{Z}$ ($[-M, 0[\cap\mathbb{Z}$ goes vice versa). First we initialize $S[x]$ with zeroes, for $0 \leq x < M$. Now, for every prime p_r in the factorbase FB, and for both $x_{r,i}, i = 1, 2$, we add $\log(p_r)$ to every location $S[x]$ where $x \equiv x_{r,i} \bmod p_r$. Starting with $x = x_{0,1} \in [0, p_0[$, we continuously increment $S[x]$ by $\log(p_0)$ and x by p_0, until we have traversed the sieve. After the sieving is finished, the value of $S[x]$ gives a rough estimate of the amount $Q(x)$ is divisible by the primes in FB. The entry of $S[x]$ is the logarithm of the product of all primes in FB dividing $Q(x)$, observe that we have ignored powers of primes. We scan through the sieve and compare every entry $S[x]$ with a certain threshold $T = \log(\sqrt{2N}M) - \delta$ which represents approximately the logarithm of $Q(x)$ reduced by a small fudge factor δ. If $S[x]$ is bigger than T, $Q(x)$ will be computed and divided over the primes in FB. Here we make again use of the fact that p divides $Q(x)$ if and only if $x \equiv x_{r,1}$ or $x_{r,2} \bmod p_r$. After at least $R + 1$ relations $Q(x) \equiv z_x^2 \bmod N$ are collected, in a second step a subset among them is searched such that by multiplying these relations we receive a perfect square on the left side, too. With $X^2 \equiv Y^2 \bmod N$, X and Y being "enough" random, there is a good chance that $\gcd(X - Y, N)$ is a nontrivial divisor of N.

Taking into account the hypercube variation of QS and the published literature, it is safe to say that when factoring 100 decimal digit numbers, 70% of the running time is spent on sieving, about 10% on the evaluation of the sieve and 20% on computing the values of $Q(x)$ and trial dividing over the primes in FB. With growing N the proportion of the sieving time will increase, because the number of candidates x out of the sieve decreases.

Now we will roughly describe the architecture of a modern computer as it is relevant for us. For a complete description, see [4, 5, 9], for example. The main ingredients concerning memory architecture important for sieving are the number of the caches, its sizes and types, and size and type of the Translation Lookaside Buffer (TLB). We assume a first level cache of respectable size (typically 32KB or more), either unified (because there are few instructions needed for sieving only) or separated (following the harvard architecture). The Translation Lookaside Buffer mainly manages memory pages, whose' size and the number of entries describes the amount of memory it can handle at the same time, which we will denote by s_T. Let s_p denote the size of a memory page and s_l the size of a cache line. In our simplified model data access occurs as follows. First the CPU checks wether the actual line (where the data belongs to) is in the first level cache or not. In the latter case the TLB is checked wether it contains the physical page address of the actual page (where the data belongs to), this is typically done in parallel. In the case of a TLB miss, the physical page address must be computed, and in the worst case of a page fault (which we do not consider here) the page must be read from an external device. By $t_l(.)$ we mean the time in clock cycles (the *access time*) needed to load a given data byte or word into a register, depending on a first level cache hit (1), a first level cache miss, but a TLB hit (2), and a TLB miss (3). These values ($t_l(1) : t_l(2) : t_l(3)$) may be as bad as 1–2 : 5–10 : 10–40, and even worse [4, 5]. Note that we cannot exploit a kind of burst mode because we do not acces a stream of data bytes here (execpt during initialization and evaluation of the sieve). The omission of a first level cache, and/or the inclusion of a second level cache in our model is quite easy and straightforward. A similiar kind of analysis should be done for write accesses, depending on write-back or write-through caches, the existence and size of writebuffers and much more. We will ignore these aspects completely here.

```
1     initialize(S, M)
2     for (r = r_0; r < R; r = r + 1) do
3           d_1 = x_{r,2}; d_2 = p_r - d_1; M_p = M - d_1
4           for (x = x_{r,1}; x < M_p;    ) do
5                 S[x] = S[x] + l_r
6                 x = x + d_1
7                 S[x] = S[x] + l_r
8                 x = x + d_2
9           endfor
10          if (x < M) then
11                S[x] = S[x] + l_r
12    endfor
13    evaluate(S, M)
```

Figure 1: Naive algorithm

2 Variations on the Sieve

In this section we restrict ourselves to the sieve interval $[0, M[$, the interval $[-M, 0[$ goes vice versa. We concentrate on the sieving part and skip both initialization and evaluation of the sieve where the sieve is accessed from bottom to top one word after another.

Let M be the sievelength and R the size of the factorbase. Let $FB = \{p_r : 0 \leq r < R\}$ the factorbase and $0 \leq x_{r,1}, x_{r,2} < p_r$ the roots of the actual polynomial $Q(X)$ modulo p_r, for $0 \leq r < R$. Let r_0 be the index of the first prime bigger than the small prime bound, which means we skip sieving for the primes $p_r, r < r_0$. We assume $x_{r,1} \leq x_{r,2}$ and store the difference in $x_{r,2}$ rather than $x_{r,2}$ itself (this resembles a loop-unrolling of depth two). Let $l_r, 0 \leq r < R$ be a precomputed table of the rounded logarithms of the primes in FB. (In practice one uses a compact representation of l_r, because the entries grow very slowly. For the sake of simplicity we describe the algorithms with the ordinary representation.)

The naive approach sieves the whole interval at once (see Figure 1).

The most time-consuming part of the naive algorithm is the access of the elements $S[x]$ in lines 5 and 7 (ignoring the single access in line 11). If M is large enough we can assume a cache miss every $\lceil s_l/p_r \rceil$-th access and a TLB miss every $\lceil s_p/p_r \rceil$-th access. Since the size s_l of a cacheline is typically very small (say, 128 bytes), there is a cache miss in most of the cases. Compared with the sievelength M the pagesize s_p is typically very small, too (4KB). Every prime p_r causes $2M/p_r$ accesses. We estimate the complexity roughly (ignoring the accesses of p_r, l_r and $x_{1,r}, x_{2,r}$, initialization and evaluation of the sieve) and a bit too pessimistic by the formulae

$$5(R - r_0) + \sum_{r_0}^{R} M/p_r \, (3 + 2t_l(3)) \tag{1}$$

where $t_l(3)$ is the time (in clock cycles) needed for a load with TLB miss. Additionally to an extensive $t_l(3)$, it may not be possible to have enough physical RAM to hold the whole interval $[0, M[$.

The quite natural approach hence sieves the interval $[0, M[$ blockwise with blocks of size B:

$$[0, B[, [B, 2B[, \ldots, [(k-1)B, M[$$

where $k = \lceil M/B \rceil$. After the first block $[0, B[$ is sieved, we have to adjust the modular roots $x_{r,1}, x_{r,2}$ relative to the starting point B of the second block. The primes are divided into two classes. Let $p_r, r_0 \leq r < r_s$ be the small primes such that p_{r_s} is the first prime bigger than the blocksize B. Hence, the modular roots of the small primes always lie inside the block. The rest of the primes in FB are the big primes and have at most two modular roots inside the block. For simplicity let $B|M$, the whole algorithm then looks as follows (see Figure 2).

Since the time for the initialization and the evaluation of the sieve is linear in the sievelength we don't expect an overhead in the single block algorithm for the initialization and the evaluation of the sieve. Assuming $B \leq s_T$, no TLB miss occurs. We roughly estimate the work for the single block algorithm by

$$3(R - r_0) + \sum_{r_0}^{R} M/p_r \, (3 + 2t_l(.)) + k \, (3(R - r_s) + 7.5(r_s - r_0)) \tag{2}$$

where $t_l(.)$ is either the time $t_l(1)$ needed for a load with cache hit (if B is not greater than the size of the first level cache) or the time $t_l(2)$ needed for a load with TLB hit (but cache miss). The optimal blocksize B obviously depends on the machine used, and is best determined experimentally.

Because the ratio $t_l(2)/t_l(1)$ is on most machines even worse than the ratio $t_l(3)/t_l(2)$, and because the smallest primes are the most expensive ones, we search for another modification which makes effective use of the the first level data cache.

We use two blocks of size B_1 (the inner block) and of size $B_2 \geq B_1$ (the outer block). The primes are divided into four classes. Let $p_r, r_0 \leq r < r_s$ be the small primes such that p_{r_s} is the first prime bigger than B_1. Hence, the modular roots of the small primes always lie inside the inner block. Let $p_r, r_s \leq r < r_m$ be the medium primes. The optimal choice of r_m will be explained later. Take, for example, p_{r_m} as the first prime bigger than $B_2/4$. Finally, let $p_r, r_m \leq r \leq r_l$ be the set of large primes such that p_{r_l} is the first prime bigger than the size of the outer block B_2, and $p_r, r_l \leq r < R$ be the remaining (extra large) primes.

Now the interval $[0, M[$ will be initialized and evaluated in outer blocks of size B_2 as in the single block algorithm. Every outer block of size B_2 becomes sieved in inner blocks of size B_1 by the small and medium primes. Then it will be sieved by the large and extra large primes as before. The medium primes should be chosen such that the bookkeeping overhead from rewriting the modular roots relative to the starting point of the inner block is still compensated by the sieve accesses, which are first level cache hits here. We assume $B_1|B_2$ and $B_2|M$ with $k_2 = M/B_2$ and $k_1 = B_2/B_1$. A sketch of the whole algorithm then reads as follows (see Figure 3).

Every access of $S[x]$ during the b_1-loop in line 4 results in a first level cache hit. Every access of $S[x]$ in line 6 results in a TLB hit as in the single block algorithm. Again, we roughly estimate

```
1   for (b = 0; b < k; b = b + 1) do
2       initialize(S, B)
3       for (r = r_0; r < r_s; r = r + 1) do
4           d_1 = x_{r,2}; d_2 = p_r − d_1; B_p = B − d_1
5           for (x = x_{r,1}; x < B_p;    ) do
6               S[x] = S[x] + l_r
7               x = x + d_1
8               S[x] = S[x] + l_r
9               x = x + d_2
10          endfor
11          if (x < B) then
12              S[x] = S[x] + l_r
13              x = x + d_1
14              x_{r,2} = d_2
15          else
16              x_{r,2} = d_1
17          endif
18          x_{r,1} = x − B
19      endfor
20      for (r = r_s; r < R; r = r + 1) do
21          if ((x = x_{r,1}) < B) do
22              S[x] = S[x] + l_r
23              x = x + x_{r,2}
24              if (x < B) do
25                  S[x] = S[x] + l_r
26                  x = x + p_r − x_{r,2}
27              else
28                  x_{r,2} = p_r − x_{r,2}
29              endif
30              x_{r,1} = x − B
31          else
32              x_{r,1} = x_{r,1} − B
33          endif
34      endfor
35      evaluate(S, B)
36  endfor
```

Figure 2: Single block algorithm

```
1     for (b_2 = 0; b_2 < k_2; b_2 = b_2 + 1) do
2           initialize(S, B_2)
3           for (b_1 = 0; b_1 < k_1; b_1 = b_1 + 1) do
4                 sieve [b_1 B_1, (b_1 + 1) B_1] using the single block algorithm and
                  primes up to indices r_s and r_m (without initialization and evaluation)
5           endfor
6           sieve [b_2 B_2, (b_2 + 1) B_2] using the single block algorithm and
            primes with indices from r_m to r_l and R (without initialization and evaluation)
7           evaluate(S, B_2)
8     endfor b_2
```

Figure 3: Double block algorithm

the whole work by

$$\begin{array}{l} \sum_{r_0}^{r_m} M/p_r\,(3+2t_l(1)) + \sum_{r_m}^{R} M/p_r\,(3+2t_l(2)) \\ +k_2(3(R-r_l)+7.5(r_l-r_m)+k_1\,(3(r_m-r_s)+7.5(r_s-r_0))) \end{array} \tag{3}$$

The optimal choice of r_m of course depends on the machine used and is best determined experimentally.

Remark. Register use, the direct access of S[x] and the scheduling of load and store instructions are crucial steps, too. Probably, the compiler's assembler output should be verified.

3 An Example

Here we will verify our theoretical considerations of the last section. While the double block algorithm works efficiently already when the biggest prime in the factorbase exceeds the cache size, we choose an 116 decimal digit composite as an example. The number N is a remaining cofactor of $7^{194}+1$ of the Cunningham project [3]. We chose $R = 200.000$ and $M = 32 \cdot 2^{20}$. The small multiplier (see Section 4) used is five, which results in a factorbase consisting of all primes p less or equal than 5797439 such that $5N$ is a quadratic residue modulo p. Using a small prime bound of 70, r_0 equals 10. The actual machine is an IBM RS/6000 250 workstation equipped with an 66Mhz PowerPC-601 processor (without second level cache) and 64MB RAM running under AIX 3.2.5. The first level unified instruction and data cache is 32KB big, 8-way set-associative with LRU replacement, with lines consisting of two sectors of 32 bytes each. The TLB has 128 entries and is 2-way set-associative, the page size is 4KB. Therefore we choose a blocksize of 512KB (128·4KB) for the single block algorithm (which results in $r_s = 21796$), and blocksizes of 32KB and 512KB for the double block algorithm. (Even after extensive experimenting with other sizes, these values still have been best.) As medium sized primes for the double block algorithm we take all primes less than $128 \cdot 2^{10}$, from which the values 1743, 6887 and 21796 of r_s, r_m and r_l follow.

algorithm	time (in sec)	sieve speedup	total speedup
naive	146,3	1,0	1,0
single block	107,6	1,36	1,32
double block	55,3	2,64	2,27

Table 1: Practical results

We assume a ratio of $1:5:10$ for $t_l(1):t_l(2):t_l(3)$. Computing only the number of sieve access operations according to equations 1, 2 and 3 yields a speedup of 2,0 for the single block algorithm and of 5,21 for the double block algorithm over the naive algorithm. Taking into account the overhead of the bookkeeping operations, the predicted speedup over the naive algorithm is 1,43 for the single block and 1,97 for the double block algorithm. Table 1 contains the times needed for sieving one polynomial on the IBM RS/6000 250 workstation (ANSI-C, xlc 1.3.0.23 optimized with -O3). The achieved speedup over the naive algorithm is 1,36 for the single block and 2,64(!) for the double block algorithm. Spending additional 16,3 seconds for computing the values of $Q(x)$ and trial dividing over the primes in the factorbase, the total speedup is 1,32 for the single block algorithm and 2,27 for the double block algorithm.

4 More Remarks

In this section we will summarize previous methods to speed up the sieving phase of QS (for the sake of completeness), and discuss the application of the results of section 2 to NFS.

Previous additional methods to speed up the sieving phase of QS (these were already incorporated in our implementation):

- sieve initialization with logarithms of small primes [10]
- initialization and evaluation using machine words [10]
- skip sieving for small primes and prime powers [7, 10], respecitively
- sieve true powers of small primes instead of small primes [1]
- use of a second threshold during the trial division [1]

In the sieving phase of the NFS algorithm [2] we are looking for pairs (a,b) of integers with $a \neq 0, b > 0, \gcd(a,b) = 1$, such that (for a given number m and a polynomial f) $a - bm$ factors completely over a first (rational) factorbase FB_1, and $N(a,-b) = b^d f(a/b)$ factors completely over a second (algebraic) factorbase FB_2. There are several possibilities of sieving the rectangle $([-M_a, M_a[\times[1, M_b[) \cap \mathbb{Z}^2$. First, one can fix either an a-value (or an b-value) and sieve one line $[1, M_b[$ (or $[-M_a, M_a]$), first according to $a - bm$ and then according to $N(a,-b)$ (or vice versa). Here we can apply the double block algorithm immediately. A more improved method of sieving is due to J. M. Pollard and is called the lattice sieve [6]. Basically the idea is the following. Fix

a medium size prime q. Instead of sieving the whole rectangle from above, we restrict ourselves to the numbers (a, b) such that q divides $a - bm$, which form a lattice L_q. Let v_1, v_2 be a basis of short vectors of L_q, and let $([-M_c, M_c[\times[1, M_d[) \cap \mathbb{Z}^2$ be the parameter rectangle for v_1 and v_2. So we are looking at the points $cv_1 + dv_2$ with $-M_c \leq c < M_c$ and $1 \leq d < M_d$. Now in order to sieve L_q with a prime p either in FB_1 or in FB_2, Pollard points out that the pairs (a, b) such that p divides $a - bm$ form a sublattice L_{pq} of L_q. He suggests sieving L_q by rows for the small primes in FB_1 and FB_2, which means fixing an c-value and sieving $c \cdot v_1 + [1, M_d[\cdot v_2$ (or vice versa). Here the double block algorithm applies, too. For larger primes Pollard suggests sieving by vectors, which means computing a basis of L_{pq}. Here we do not see any possibility to apply the double block algorithm; on the other hand, it does not seem to make sense, because the primes are large.

References

[1] D. Atkins, M. Graff, A. K. Lenstra, P. C. Leyland, "THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE", *Proceedings ASIACRYPT '94*, to appear.

[2] J. P. Buhler, H. W. Lenstra, Jr., Carl Pomerance, "Factoring integers with the number field sieve", in: A. K. Lenstra, H. W. Lenstra, Jr. (Eds.), *The development of the number field sieve*. Lecture Notes in Mathematics 1554, 1993, 50-94.

[3] J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman, and S. S. Wagstaff, Jr., *Factorizations of $b^n \pm 1$ for b = 2, 3, 5, 6, 7, 10, 12, up to High Powers*. American Mathematical Society, Providence, Rhode Island, 1983.

[4] Kevin Dowd, *High performance computing*. RISC architectures, optimization, & benchmarks. O'Reilly, Sebastopol, CA, 1993.

[5] J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[6] J. M. Pollard, "The Lattice Sieve", in: A. K. Lenstra, H. W. Lenstra, Jr. (Eds.), *The development of the number field sieve*. Lecture Notes in Mathematics 1554, 1993, 43-49.

[7] C. Pomerance, "The Quadratic Sieve Factoring Algorithm", *Advances in Cryptology, Eurocrypt '84*, Lecture Notes in Computer Science 209 (1985), pp.169-182.

[8] C. Pomerance, "Factoring", in: *C. Pomerance (ed.), Cryptology and Computational Number Theory*, Proc. of Symp. in Appl. Math. Vol.42 (1989), AMS, pp.27-47.

[9] S. Weiss, J. E. Smith, *IBM Power and PowerPC: principles, architecture, implementation.* Morgan Kaufmann, San Mateo, Calif., 1993.

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