CERIAS Tech Report 2001-45

Implementing the Hypercube Quadriatic Sieve with Two Large Primes

Brian Carrier, Samuel S. Wagstaff, Jr.
Center for Education and Research in
Information Assurance and Security
&
Department of Computer Sciences, Purdue University
West Lafayette, IN 47907

Implementing the Hypercube Quadratic Sieve with Two Large Primes

Brian Carrier and Samuel S. Wagstaff, Jr.*
Center for Education and Research
in Information Assurance and Security
and Department of Computer Sciences, Purdue University
West Lafayette, IN 47907-1398 USA

Abstract

This paper deals with variations of the Quadratic Sieve integer factoring algorithm. We describe what we believe is the first implementation of the Hypercube Multiple Polynomial Quadratic Sieve with two large primes, We have used this program to factor many integers with up to 116 digits. Our program appears to be many times faster than the (non-hypercube) Multiple Polynomial Quadratic Sieve with two large primes.

Key Words: Quadratic Sieve, Hypercube Quadratic Sieve, Self-Initializing Quadratic Sieve, Multiple Polynomial Quadratic Sieve, Two Large Primes.

1 Introduction

The Quadratic Sieve integer factoring algorithm (QS) was initially proposed by Pomerance in 1982 [13]. Since then, improvements made to the algorithm include the Multiple Polynomial Quadratic Sieve (MPQS) [5, 15], the Hypercube Multiple Polynomial Quadratic Sieve (HMPQS) or the Self Initializing Quadratic Sieve [1, 12], the Quadratic Sieve with one large prime [6], and the Quadratic Sieve with two large primes [9]. Computer programs have been written for the MPQS with one and two large primes and the HMPQS with one large prime. So far as we know, we are the first to implement the

* This work was supported in part by the Purdue University CERIAS and by the Lilly Endowment, Inc. The authors' email addresses are carrier@cerias.purdue.edu and ssw@cerias.purdue.edu

HMPQS with two large primes. Even if our implementation is not the first one, we have introduced a novel method of choosing polynomial coefficients, which makes them closer to the ideal values than the approximations used in earlier HPMQS programs, and accelerates the program. Another new feature of our program is the automatic adjustment of one parameter (the "sieve threshold") to move it closer to its optimal value while the program runs. So far, we have factored many numbers of modest size and a few larger ones and have seen excellent performance.

The second section of this paper introduces notation and gives a brief overview of the original Quadratic Sieve and its development. The third section describes the new features of our program and the fourth section presents some results we observed while factoring numbers for the Cunningham Project [2].

2 Overview

This section describes the Quadratic Sieve and gives a brief summary of its history and its variations.

2.1 Single Polynomial

The Quadratic Sieve factors an odd positive integer n, not a prime power, by finding a solution to $x^2 \equiv y^2 \pmod{n}$ with $x \not\equiv \pm y \pmod{n}$ for $x, y \in Z$. The first congruence implies that n divides $(x-y)(x+y)$, while the second congruence implies that n divides neither $(x-y)$ nor $(x+y)$. Therefore, one prime factor of n divides $(x-y)$ but not $(x+y)$ and another prime factor of n divides $(x+y)$ but not $(x-y)$. It follows that the greatest common divisors $(n, x-y)$ and $(n, x+y)$ will be proper divisors of n.

Since there is no easy method of finding a nontrivial solution to $x^2 \equiv y^2 \pmod{n}$, the QS finds many relations of the form $z_i^2 \equiv q_i \pmod{n}$, where the prime factorization of q_i is known. The prime divisors of q_i are from a set of primes called the Factor Base (FB). Usually, these are the first primes for which n is a quadratic residue. When we have found more relations than there are primes in the FB, the relations can be combined using linear algebra over the field GF(2). The relations are combined so that the product of the q_i values is a square, say y^2, and then the desired relation $x^2 \equiv y^2 \pmod{n}$ is formed, where x is the product of the corresponding z_i's. This process is described in [14].

In the initial design [13, 6] the single polynomial $f(x) = (a+x)^2 - n$ was used, where $a = \lfloor \sqrt{n} \rfloor$. If we let $z = (a+x)$ and $q = f(x)$, then the

desired relations $z_i^2 \equiv q_i \pmod{n}$ are produced whenever we find an x value such that $f(x)$ can be factored by the primes in the *FB*. To determine which values of x will yield such an $f(x)$, the zeros of $f(x)$ are calculated modulo all primes in the *FB*. A block of consecutive values of x is represented by an array of bytes. The array is initialized to 0 and for every x that is a zero of $f(x) \pmod{p}$, an integer approximation to $\log(p)$ is added to the corresponding array location. This process is called a *sieve* because of its resemblance to the Sieve of Eratosthenes. After the polynomial has been sieved for all primes in the *FB*, the array is scanned for values that exceed a specified threshold. These locations correspond to the values of x where $f(x)$ has many prime divisors in *FB*; these are examined further by trial division.

The trial division phase begins by evaluating $f(x)$ for the x values discovered during the scan and dividing them by the primes in the *FB*. The division can be done efficiently if one notes that the only divisors of $f(x)$ are those primes p for which $x \equiv r \pmod{p}$, where r is one of the zeros of $f(x) \pmod{p}$. If $f(x)$ can be completely factored with primes in the *FB*, then it is called a *full relation* and it is saved in a file. In this primitive version of the QS, the sieving process continues until there are more full relations than primes in the *FB*.

2.2 Large Primes

When the trial division method just described is used, the algorithm does not save values of x when $f(x)$ does not factor completely using primes in the Factor Base. The size of the *FB* would have to be quite large for this to work well and the sieving process would take a long time. This problem was solved already in the Continued Fraction Algorithm, an earlier integer factoring algorithm. Morrison and Brillhart [10] proposed saving the relations that have at most one prime factor larger than the largest prime F in the factor base and smaller than some upper bound P. This technique has been used in every implementation of the Quadratic Sieve, even the first one [6]. These relations with one prime beyond F are called *partial relations*. The partial relations are stored, and any two partial relations containing the same large prime can be multiplied, and the common large prime removed, to form a full relation. It takes no extra effort to find partial relations when $P \leq F^2$ since we know that the remaining cofactor is prime because the trial division has already searched for all possible prime divisors below its square root.

Another variation, due to A. K. Lenstra and Manasse [9], saves relations

that have at most two large primes less than P and greater than F. This method takes a small additional effort, since the cofactor remaining after trial division may have to be factored into the two large primes. Because the remaining cofactor could also be a single large prime, a probable prime test is performed to distinguish prime and composite cofactors so that factorization is attempted only for the composite ones. Relations that contain two large primes are called *partial-partial relations*, or *pp*'s, and can be combined with partial and other *pp* relations by a graph cycle-finding algorithm to form full relations [9].

2.3 Multiple Polynomials

As the size of x in the sieve increases, the probability of getting a full relation decreases. It was proposed by Davis and Holdridge [5] and Montgomery [14] to use many polynomials for shorter sieve intervals. The new Multiple Polynomial Quadratic Sieve (MPQS) is significantly faster than the single polynomial version but requires expensive multi-precision and modular inverse operations for each new polynomial. The algorithm spends much time calculating the new zeros for each polynomial when compared to the sieving time.

The next version of the algorithm was invented independently by Peralta [12] and Alford and Pomerance [1], who respectively called it the Hypercube Multiple Polynomial Quadratic Sieve (HMPQS) and the Self-Initializing Quadratic Sieve. This paper will refer to the algorithm as the HMPQS. The algorithm uses polynomials with two coefficients, a and b, of the form $f(x) = (ax + b)^2 - n$. The details of this polynomial construction will be given in the implementation section, but its overview is:

- a has s prime factors $q_1 \ldots q_s$, where $q_i \in FB$, $1 \leq i \leq s$.
- b is the sum of s values.
- $b^2 \equiv n \pmod{a}$. There exist 2^s solutions to this equation, but only 2^{s-1} are of interest because the other half represent the negative values of the first 2^{s-1} values, and would yield duplicate relations.
- The 2^{s-1} values of b and corresponding zeros of $f(x) \pmod{p}$ are quickly computed with a Gray code using single precision arithmetic instead of the multi-precision arithmetic needed in the original MPQS.

The sieving and trial division process of the HMPQS is the same as with the

single polynomial QS, except that the HMPQS does not sieve by the prime factors of a.

3 Implementation Details

This section describes the details and new features incorporated in our program.

3.1 A-Generation

The a coefficient in the hypercube polynomial $(ax+b)^2 - n$ is the product of s primes, $q_1 \ldots q_s$, from the factor base and has an ideal value of $\sqrt{2n}/M$, where M is the length of the sieve array. The method of choosing the prime factors of a in our implementation is due to the second author. The first step in calculating a is to calculate the logarithm of the ideal value of a. The prime factors of a are divided into three sets: $\{q_1, q_2, q_3\}$, $\{q_4, q_5, \ldots q_{s-3}\}$, and $\{q_{s-2}, q_{s-1}, q_s\}$. The FB is also divided into three sets starting from an offset in order to avoid small primes. The first set of prime factors, $\{q_1, q_2, q_3\}$, is selected upon program initialization based on a given machine number. These primes are taken from the first `MACH_BND` valid primes in the FB, where we chose `MACH_BND` to be 35. This allowed many machines to sieve on the same number without duplicating relations. The logarithm of the product of the machine-based primes is then subtracted from the ideal logarithm of a. The value of s is determined by dividing the remaining logarithm by the logarithm of the prime in the FB at index `MACH_BND` + `A_BND`/2 and adding 3, where we chose `A_BND` to be 200. This part of the A-Generation algorithm was present already in the programs of Contini and Peralta.

The middle set of primes is taken from the primes with odd indices in the FB within the range of `MACH_BND` + 1 to `MACH_BND` + `A_BND`, The primes in this set are selected in lexicographic order using the "Next k-subset of an n-set" algorithm `NEXKSB` in Nijenhuis and Wilf [11]. For each new a, the next $(s-6)$-subset of these `A_BND`/2 primes are used.

The final three primes, $\{q_{s-2}, q_{s-1}, q_s\}$, are taken from the even indices within the same range as the middle primes. When the sieve program starts, a lookup table is created with products of all combinations of three even-indexed primes in the above range. The logarithm of that combination serves as the index to the lookup table. Each time a new set of middle primes is chosen, the lookup table is used to determine which sets of three primes can be used to produce the ideal value of a to within a certain tolerance, say

0.01%. In practice however, the range of the middle primes was modified to be more efficient. The lower bound was set such that the logarithm of the product of the first combinatorial set of primes plus the largest logarithm in the lookup table would produce the logarithm of the ideal a. Similarly, the upper bound for the middle primes was set so that the logarithm of the product of the last set of primes plus the smallest logarithm from the lookup table produced the logarithm of the ideal a. This avoids having to cycle through sets of middle primes that require a value that is larger or smaller than the largest or smallest combination in the lookup table. However, this method does not guarantee that the size of the middle range will be `A_BND`. This strategy produces a values much closer to the ideal value than the programs of Contini and Peralta produced, and results in better performance for our program.

Table 1 shows the number of hypercubes and polynomials that can be sieved with for a given machine number. A 60-digit number uses one hypercube every 1 to 2 minutes and if we conservatively use 1 minute, then it will take over 7 days before a new machine number is needed. Since a 60-digit number is factored in less than 5 minutes, this is not a problem. Similarly, sieving for the 116-digit number takes about 8.5 hours per hypercube, so a given machine number will not run out of primes until after 20,836 years of continuous sieving, and the number would be factored long before this time.

Size in Digits	Hypercubes	Polynomials per Hypercube	Total Polynomials
60	10296	128	1317888
70	3183545	512	1629975040
80	16108764	1024	16495374336
90	99884400	4096	409126502400
100	95548245	8192	782731223040
110	92561040	32768	3033040158720
116	21474180	65536	1407331860480

Table 1: Number of Hypercubes per Machine Number

This table assumes that each middle set of primes has exactly one corresponding set of final three primes in the lookup table, and does not take into account the instances where there are zero or several final three primes that will produce the ideal value of a. It should also be noted, that the number of hypercubes starts to decrease at about 90 digits. This is because the difference between the upper and lower bounds of the middle primes decreases

as n increases, due to the optimization mentioned above. If this should become a problem for larger numbers, it can be solved easily by increasing the bounds of the lookup table.

3.2 B-Generation

This part of the algorithm is based on the work of Contini and Peralta. The b coefficient of $(ax+b)^2 - n$ must satisfy $b^2 \equiv n \pmod{a}$, and be the sum of s values B_i, $1 \leq i \leq s$, where s is the number of prime divisors of a. The values of B_i are combined using a Gray code, for example $b_1 = B_1 + B_2 + B_3 + \cdots + B_{s-1} + B_s$ and $b_2 = B_1 + B_2 + B_3 + \cdots + B_{s-1} - B_s$. If each B_i satisfies $B_i^2 \equiv n \pmod{q_i}$, where the q_i are the prime divisors of a and $B_i \equiv 0 \pmod{q_j}$ for all $i \neq j$, then the square of the sum of the B_i's will be congruent to $n \pmod{a}$ by the Chinese Remainder Theorem. The formulas we used to calculate the values of B_i are taken from [3] or [4]:

$$t_{q_i}^2 \equiv n \pmod{q_i}$$

$$\gamma_{a,q_i} \equiv t_{q_i} \left(\frac{a}{q_i}\right)^{-1} \pmod{q_i}$$

$$B_i = \frac{a}{q_i}\gamma_{a,q_i}$$

There are two choices for t_{q_i} and the one that makes γ_{a,q_i} smaller is used.

If n is multiplied by a small number m to create a more favorable factor base, then care must be taken to ensure that a is relatively prime to m. Otherwise, if $q_i | a$ and $q_i | m$, then $B_i^2 \equiv 0 \equiv mn \pmod{q_i}$ and the Gray code will only cycle through half of the b_i values, while generating each polynomial twice and therefore outputting each relation twice.

During initialization, we calculated the square root of n modulo p for each prime in the factor base and stored the value in the factor base file. Therefore, finding the value of t_{q_i} is an array lookup. If the value of γ_{a,q_i} is greater than $q_i/2$, then it was subtracted from q_i, which is equivalent to using the other value of t_{q_i}.

As above, let $b_1 = B_1 + B_2 + B_3 + \cdots + B_s$. We can compute the b_j's iteratively using a Gray code:

$$b_{j+1} = b_j + 2(-1)^{\lceil j/2^v \rceil} B_v$$

where $2^v || 2j$ ("exactly divides") for $j = 1, \ldots, 2^{s-1} - 1$. Since the change in b_j to get b_{j+1} is $\pm 2B_i$, the program saves the values of $2B_i$ during the

computation of b_1. The formula for calculating the zeros of the polynomial is similar:

$$r_{j+1,p} \equiv \pm \left(r_{j,p} - 2(-1)^{\lceil j/2^v \rceil} B_v a^{-1} \right) \pmod{p}$$

where again $2^v || 2j$. The zeros are needed for each prime $p \in FB$, so the values of $B_i a^{-1} \pmod{p}$ are computed and stored in memory when the values of B_i are determined. Therefore, a new polynomial requires a simple loop to determine the highest power of 2 that divides $2j$:

```
for (i=1; (j & (0x1 << (i-1) )) == 0; i++) {}
```

and $2 \times \#FB + 1$ single precision additions or subtractions.

3.3 Sieve Scan

The sieve was scanned swiftly by a trick of Contini [3]. Instead of initializing the sieve to 0 and checking each value to see if it exceeded a given threshold `T`, the byte-sized sieve values were initialized to `0x80 - T`. The scanning process consisted of a bitwise `AND` of a four-byte word of the sieve with `0x80808080`. If the `AND` were non-zero, it meant that at least one of those 4 bytes had a value greater than or equal to `0x80`. In this case, the bytes were examined individually and trial division was performed on any hits.

3.4 Redundant Restart Files

Due to past experience of the second author, our program had redundant restart files. They are useful because, if only one restart file were used and the program terminates while writing to it, no restart information at all would be stored. This failure actually occurred daily on one flaky supercomputer we used, and the problem used to require manual correction.

In our program, two restart files are written alternately. When the program is restarted, it begins with the largest restart value found in a valid restart file. Our program saves the index to the combinatorial algorithm `NEXKSB` that selects the middle prime factors of a as the restart value. In other words, we only save restart information for each set of hypercubes with common middle primes, not for each polynomial or even each hypercube.

3.5 Dynamic Sieve Settings

The program used a file that allowed the operator to change the sieve threshold for the next hypercube while the program was running. It also allowed the operator to tell the program to stop sieving after the next hypercube has finished. This allows the operator to get the maximum number of relations

from the program by choosing an optimal threshold and using an entire hypercube before stopping. The QS program for the MasPar of A. K. Lenstra [7] had some similar features.

3.6 Auto Adjust Sieve Threshold

Another novel feature of our program, due to the first author, was logic to find the optimal sieve threshold. The feature avoids the tedious trial and error process of finding the optimal threshold setting, which can vary among numbers of the same length. The program assumes that there is only one optimal threshold value, which has not been proven. The program counts how many partial and *pp* relations are found from each hypercube for at least two minutes and calculates the number of partials per second and *pp*'s per second. It then calculates a weighted sum, since partials are more important than *pp*'s. The sieve threshold is then changed by one, and the process continues until the program finds a threshold that has smaller weighted sums above and below it. The full relations are not counted in this optimization because they are so rare and occur irregularly.

This feature is still being researched to determine what weights should be given to the relations. We obtained good results for a 76-digit number when we gave the partials weight 5 and the *pp*'s weight 1. However, a 76-digit number is so small that more partial relations than *pp* relations are generated. More tests with larger numbers should be conducted to determine good weights.

4 Results

This section will give the results we observed while writing this program and compare them to the results of a program for MPQS with two large primes.

4.1 Parameters and Times

This program was used to factor more than 110 numbers for the Cunningham project. From these experiences, Table 2 gives some of the parameters we used and times we observed.

Only one 92 digit number was factored, so the parameters for it may not be optimal. The times are those needed when the program was run on a Sun workstation.

Our program also kept track of how many trial divisions were performed and reported a ratio corresponding to the number of trial divides that were

Size in Digits	Factor Base Size	Sieve Length	Ave. Time
60	8000	50000	3 Mins
70	12000	50000	21 Mins
92	30000	300000	3000 Mins

Table 2: Parameter Settings

performed which produced no relation, a miss, versus the number of relations found. Table 3 gives the misses-per-relation results that we observed.

Size in Digits	Misses per Relation
60	0.03
70	0.77
80	3.38
91	4.98
116	54.3

Table 3: Misses per Relation Ratio

4.2 HMPQS versus MPQS

We have tried to compare the speed of the HMPQS and MPQS algorithms for factoring large integers. When two algorithms each take a long time to accomplish a common goal, it can take a long time to compare them. To make a fair comparison, (a) common parts of the programs for the two algorithms should be the same, (b) the same machine should be used for comparison runs, and (c) the parameters should be chosen optimally for each program.

The MPQS program we used was written by Silverman (the two large prime version of the one reported in [15]). Our HMPQS program was developed using Silverman's program as a base. Both programs used essentially the same sieving method and relation-harvesting code. The two programs differed only in how polynomials are generated and in the minor changes described in Sections 3.4-3.6 above. (The sieve scanning trick in Section 3.3 was added to the MPQS program before any of the experiments described in this paper were done.) All testing was done on one of four processors of a Sun Enterprise E4500/E5500 having 1 GB of RAM and an Ecache of 4

MB. This is how we met the first two requirements for a fair comparison of the algorithms.

We considered how best to choose optimal parameters for each algorithm. The parameters we could vary were the factor base size, the length of the sieve interval and the sieve threshold. The easiest performance measure to observe was the rate of production of relations. While comparing the programs we turned off the auto adjustment of the sieve threshold and set this value manually.

Perfect testing would require completely factoring a fixed integer for each choice of parameters for each algorithm. At least we would have to generate all the required relations for each choice. This amount of work would have been too long even for the 70-digit composite integer we used in the first test. Here is how we shortened this process: If the factor base size is fixed, then the total number of relations needed to factor the number is also fixed, although somewhat fewer relations might suffice when there is a higher proportion of full ones. For each of the two programs we tried several factor base sizes. Keeping the factor base size fixed, we ran each program for a few minutes for several sieve lengths and threshold values. We determined which choice of these two parameters produced the most relations per second, assuming that there was a unique maximum rate when only one parameter varied at a time. We noted that the proportions of full relations and partial relations did not vary much during these tests. Then we ran the program once with the given factor base size and optimal sieve length and threshold until it generated enough relations to factor the number, and recorded the running time. We repeated all of the above steps for several factor base sizes until we found the optimal one for each program.

We found that for a 70-digit number, Silverman's MPQS program ran fastest with a factor base of 8K primes, a sieve interval of 400K and a threshold of 49 (in decimal). It took 122 minutes to produce enough relations to factor the 70-digit number. Our HMPQS program was fastest with a factor base of 12K primes, a sieve interval of 50K and a threshold of 54. It took 21 minutes to produce enough relations to factor the same 70-digit number. This first experiment suggests that HMPQS is about 5 or 6 times faster than MPQS.

Contini made a similar comparison in his thesis [4]. He used only one large prime in both programs and concluded that HMPQS is about twice as fast as MPQS, at least for factoring 60, 70 and 80-digit numbers. He noted that the speed of the program depends strongly on whether the sieve interval fits in cache memory. We considered whether our comparison might have this problem. The test machine we used claimed to have a cache of

4MB. All sieve intervals we tried were shorter than that. Using the optimal parameters for the MPQS, we tried breaking the sieve interval into two pieces of 200K, four pieces of 100K and eight pieces of 50K, and observed the same run time as for an unbroken sieve interval to produce the needed relations. Hence we are confident that our HMPQS program is 5 or 6 times faster than the MPQS program for 70-digit numbers.

When one factors a large number (having more than 100 digits) with some variation of the quadratic sieve with two large primes, one finds a few full relations, many partials and very many *pp*'s. The MPQS with optimal parameters produced roughly these proportions for our 70-digit test number, but HMPQS with optimal parameters produced many fulls, many partials and only a few *pp*'s. The optimal sieve threshold was so high that few candidate *pp*'s were ever considered. We were essentially running MPQS with two large primes and HMPQS with one large prime for 70-digit numbers. The harvest of relations was not slowed in the HMPQS program by the primality test and factoring of *pp*'s because few cofactors reached these parts of the harvest. Since MPQS with two large primes is known [9] to be faster than MPQS with one large prime, and presumably the same is true for HMPQS, it appears that our estimate that HMPQS is 5 to 6 times faster than MPQS for factoring 70-digit numbers is too low. We cannot reconcile our experiments with 70 digit numbers with those of Contini [4]. He used somewhat longer sieve intervals than we did for both algorithms, but this factor is insufficient to account fully for the different results.

How, then, can we compare MPQS and HMPQS with two large primes? The proportion of *pp*'s becomes significant when one factors a number of at least 100 digits. It takes weeks to factor one number of this size using a few workstations. Furthermore, it is much more difficult to estimate the total run time from the number of relations produced in a small fraction of the total run time for the case of two large primes than for one large prime. However, for about ten years we have been factoring integers using Silverman's MPQS program on a group of Sun workstations and, sometimes, other machines. We have experimented with its parameters and believe that we choose them close to their optimal values, except that the memory available for the linear algebra step has limited our factor base size to about 65K. During the past year we have factored several numbers with more than 100 digits using the new HMPQS program. We have experimented with various parameter values and feel they are not far from optimal. We found the time comparisons in Table 4 after adjusting for different numbers of workstations (and other computers):

Our collection of workstations was factoring a 116-digit number with

Size in Digits	Weeks for MPQS	Weeks for HMPQS	Speedup
102	20	3	6.7
109	50	5	10
112	90	10	9

Table 4: Weeks to Factor a Number with 5 Workstations

MPQS when the HMPQS program was completed. Encouraged by the good apparent speed of HMPQS, we switched the Suns to the new program. We had to keep the same factor base size (65K), but tried to optimize the sieve length and threshold. With the optimal choices, the HMPQS program produced relations for the 116-digit number nine times as fast as the MPQS program. We believe that the HMPQS is about 8 to 10 times as fast as the MPQS for factoring numbers in the 110 to 120-digit range.

5 Conclusion

Our new Hypercube Quadratic Sieve with two large primes program has raised the crossover point at which the General Number Field Sieve (GNFS) [8] and the QS perform equally well. QS performs better for smaller numbers while GNFS is better for larger ones. By increasing the performance of the previously best known QS program by a factor of 8 to 10 with our program, we have shown that the QS is not obsolete.

The largest number ever factored by QS was the 129-digit RSA challenge number called RSA129. A group led by D. Atkins, M. Graff, P. Leyland and A. K. Lenstra factored this number in 1994 using the MPQS with two large primes. About two years later the GNFS factored a 130-digit number called RSA130. This was done by the team of J. Cowie, M. Elkenbracht-Huizing, W. Furmanski, A. K. Lenstra, P. Montgomery, D. Weber and J. Zayer using about 10% of the total computer time used to factor RSA129 by QS. It was claimed at that time that the GNFS was ten times faster than QS for numbers of this size. Based on our work, we believe that if the HMPQS algorithm with two large primes had been used to factor RSA129, the total time would have been about one-tenth as long as for the MPQS program that was used. This would show that the crossover between QS and NFS is near 130 digits.

References

[1] W. R. Alford and C. Pomerance, "Implementing the Self-initializing Quadratic Sieve on a Distributed Network," in Number Theoretic and Algebraic Methods in Computer Science, A. van der Poorten, I. Shparlinski and H. G. Zimmer, eds., Moscow, 1993, pp. 163–174.

[2] J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman, and S. S. Wagstaff, Jr., "Factorizations of $b^n \pm 1$, $b = 2, 3, 5, 6, 7, 10, 11, 12$ up to high powers," Second edition, Contemp. Math., vol. 22, Amer. Math. Soc., Providence, RI, 1988.

[3] S. Contini, "How Fast is the Self-Initializing Quadratic Sieve?," Manuscript, February 11, 1997.

[4] S. Contini, "Factoring Integers with the Self-Initializing Quadratic Sieve," Master's thesis, University of Georgia, 1997. Available from: `http://www.maths.usyd.edu.au:8000/u/contini/`.

[5] J. A. Davis and D. B. Holdridge, "Factorization using the Quadratic Sieve algorithm," in Advances in Cryptology, Proceedings of Crypto 83, David Chaum, ed., Plenum Press, New York, 1984, pp. 103–113.

[6] J. L. Gerver, "Factoring large numbers with a Quadratic Sieve," Math. Comp. vol. 41, 1983, pp. 287–294.

[7] A. K. Lenstra, "Massively parallel computing and factoring," in Proceedings of LATIN '92, Springer-Verlag Lecture Notes in Computer Science 583, Berlin, New York, 1992, pp. 344–355.

[8] A. K. Lenstra and H. W. Lenstra, Jr., The Development of the Number Field Sieve, Springer-Verlag Lecture Notes in Mathematics 1554. Berlin, New York, 1993.

[9] A. K. Lenstra and M. S. Manasse, "Factoring with Two Large Primes," Math. Comp., vol. 63, 1994, pp. 785–798.

[10] M. A. Morrison and J. Brillhart, "A method of factoring and the factorization of F_7," Math. Comp. vol. 29, 1975, pp. 183–205.

[11] A. Nijenhuis and H. S. Wilf, Combinatorial Algorithms, Second edition, Academic Press, New York, 1978.

[12] R. Peralta, "A Quadratic Sieve on the n-Dimensional Cube," Advances in Cryptology - CRYPTO '92, Springer-Verlag Lecture Notes in Computer Science 740, 1992, pp. 324–332.

[13] C. Pomerance, "Analysis and comparison of some integer factoring algorithms," Computational Methods in Number Theory, Part 1, H. W. Lenstra, Jr. and R. Tijdeman, eds., Math. Centrum Tract 154 Amsterdam, 1982, pp. 89–139.

[14] C. Pomerance, "The quadratic sieve factoring algorithm," in Advances in Cryptology, Proceedings of EUROCRYPT 84, T. Beth, N. Cot and I. Ingemarsson, eds., Springer-Verlag Lecture Notes in Computer Science 209, 1985, pp. 169–182.

[15] R. D. Silverman, "The Multiple Polynomial Quadratic Sieve," Math. Comp. 48, 1987, 329–339.