Scott Patrick Contini
Factoring Integers with the Self-Initializing Quadratic Sieve
(Under the direction of Carl Pomerance)

In 1996, we used the self initializing quadratic sieve (siqs) to set the general purpose integer factorization record for the Cunningham project. Here, we show that this algorithm is about twice as fast as the ordinary multiple polynomial quadratic sieve (mpqs). We give running times of both algorithms for 60, 70, and 80 digit numbers. These tables show the best timings we were able to get using various parameters for each of algorithms. In all cases, the best siqs times are about twice as fast as the best mpqs times.

We also explain a way of distributing the block Lanczos algorithm to solve the matrix part of the quadratic sieve. Our method was tested on an IBM SP2 parallel computer.

Index words: factoring, quadratic sieve, lanczos, Cunningham project, self initialization

Factoring Integers with the Self-Initializing Quadratic Sieve

by

Scott Patrick Contini

B.S., Purdue University, 1992

M.S., University of Wisconsin-Milwaukee, 1994

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Master of Arts

Athens, Georgia

1997

Factoring Integers with the Self-Initializing Quadratic Sieve

by

Scott Patrick Contini

Approved:

_____ _____

Major Professor Date

Approved:

_____

Graduate Dean

_____

Date

Acknowledgements

Many of the computers used for our 116 digit factorization came from Andrew Granville's Presidential Faculty Fellows grant. Also, Red Alford, David Benson, and Carl Pomerance have generously allowed us to use their computers. Acknowledgements are also due to our system administrators, Shaheed Bacchus and Ron Rouhani, who were very helpful and also tolerant of the factoring programs. We also thank UCNS at UGA for allowing us to use their IBM SP2, and especially Alan Ferrenberg for helping us get started on the SP2.

This research has been influenced from hours of discussion with Red Alford, Andrew Granville, Arjen Lenstra, Phong Nguyen, Carl Pomerance, and Sam Wagstaff. My thesis committee of Red Alford, Andrew Granville, and Carl Pomerance were especially helpful in making the thesis more readable. I have greatly benefitted from working with Arjen Lenstra at Bellcore. Arjen has also put a lot of time into helping me with my present research. His comments and criticisms have helped me strengthen the results. I am most grateful to my advisor, Carl Pomerance. Carl has spent an enormous amount of time in helping me understand the algorithms in this thesis, as well as many other mathematical problems. Working with him has greatly improved my organization, mathematical abilities, and research skills. It amazes me that a person can spend so much time with his students and still have time to do his own research.

I cannot forget the University of Georgia Mathematics Department, which has given me an excellent education. Many of the professors here are among the best teachers I have ever had. The amount of time they have spent preparing for their

classes and helping students is very much appreciated. I thank Red Alford, Carl Pomerance, and Robert Rumely for helping get accepted to the University and the Department.

Finally, I thank my parents, who have always encouraged me to do well in academics and showed me how to be successful through hard work. I dedicate this thesis to them.

Preface

The widespread availability of computers over the past couple decades has had a great impact on mathematics. Certainly any number theorist can cite famous examples of how computers were used in proving theorems, creating new conjectures, disproving conjectures, and speeding up computation that would otherwise be very time consuming or impossible if done by hand. To the number theorist, the computer is becoming an important tool in his or her research.

But computers have also changed the way mathematicians think about solving problems. No longer are they limited to what they can do by hand, but now they can think of what a computer can do efficiently. This is reflected in how the language of mathematics has changed. For instance, when we design a new algorithm to solve a problem, we ask if it runs in "polynomial time" or if it can be easily "parallelized".

Factoring integers is perhaps the quintessential example of the importance of computers in mathematics. It is one of the oldest and most troublesome problems in mathematics. However, only in the last two or three decades has real progress been made. During this time, mathematicians have discovered some factoring algorithms that have sub-exponential running time. Moreover, their algorithms are practical: they have been used to factor integers over 100 digits long.

This thesis is about using computers for factoring. Our emphasis is on how to correctly and efficiently put these algorithms on computers. Our main interests are in the implementation of the self-initializing quadratic sieve factoring algorithm and the parallelization of the block Lanczos linear algebra algorithm, which is becoming the method of choice for solving the matrix stage of factoring algorithms. We have

also tried to highlight the most significant contributions in the evolution of the quadratic sieve. For this reason, it is somewhat expository.

Contents

Acknowledgements iii

Preface v

List of figures ix

List of tables x

1 Introduction 1

2 Quadratic Sieve 4

2.1 The Self Initializing Quadratic Sieve 9

3 Computational results 15

3.1 60 digit numbers 20

3.2 70 digit numbers 23

3.3 80 digit numbers 25

3.4 Importance of initialization 26

3.5 Remarks on factor base sizes 27

3.6 Some complete factorizations 28

3.7 Differences in factor bases 29

3.8 The factorization of a 116 digit number 31

4 The matrix problem 34

4.1 Historical solutions 36

4.2 Lanczos method 39

4.3 Applying Lanczos to factoring matrices 41

4.4 Parallel implementation of Lanczos 42

4.5 Other ideas for parallelizing iterative methods 47

5 Conclusion 49

Bibliography 50

Appendices

A QS analysis 54

B Extrapolating to 512 bits 56

C Gray codes 57

D Timings 60

E Full factorization tables 66

List of Figures

2.1 summary of basic qs algorithm 6
2.2 summary of mpqs algorithm 13
2.3 summary of siqs initialization stages 14

List of Tables

D.1 **cache experiment for first 60 digit number** 60
D.2 **siqs vs mpqs for first 60 digit number** 60
D.3 **total time to complete factorization, first 60 digit number** . . 61
D.4 **siqs vs mpqs for second 60 digit number** 61
D.5 **siqs vs mpqs for third 60 digit number** 61
D.6 **factor base experiment for first 70 digit** 62
D.7 **siqs vs mpqs for first 70 digit number** 62
D.8 **siqs vs mpqs for second 70 digit number** 63
D.9 **siqs vs mpqs for third 70 digit number** 63
D.10 **factor base experiment for first 80 digit number** 64
D.11 **siqs vs mpqs for first 80 digit number** 64
D.12 **siqs vs mpqs for second 80 digit number** 65
D.13 **siqs vs mpqs for third 80 digit number** 65

E.1 **full factorization data, first 70 digit number** 66
E.2 **full factorization data, second 70 digit number** 66
E.3 **full factorization data, third 70 digit number** 66
E.4 **full factorization data, first 80 digit number** 67
E.5 **full factorization data, second 80 digit number** 67
E.6 **full factorization data, third 80 digit number** 67

Chapter 1

Introduction

There are many factoring algorithms that are very fast for factoring integers which have small prime factors, or integers that are of a special form. These algorithms tend to be not so efficient when it comes to factoring a general integer that is a product of two primes, each of which are about the same size. However, there are algorithms that factor numbers like these in about the same amount of time as they factor any number of the same size. Such factoring algorithms are called **general purpose**, since their speed does not depend on the size of the prime factors, the number of prime factors, or the form of the number.

In practice, the best general purpose factoring algorithms known are the quadratic sieve (qs) [19] and the number field sieve (nfs) [15]. Versions of qs have been used to set all general purpose factoring records between 1984 and 1994. The largest was the factorization of the 129 digit number known as RSA-129 [2].

The asymptotically faster nfs is relatively new compared to the qs, and wasn't believed to be practical at first. It wasn't until after RSA-129 that the nfs became developed enough to factor some large integers efficiently. One of the necessary ingredients to its success was the implementation of some very fast linear algebra routines. In 1996, nfs was used to factor a 130 digit number known as RSA-130, breaking the qs record by one digit. The researchers who accomplished this claim "sieving for RSA-130 could have been done in less than 15% of the time spent on the 129 digit number." [8]

So if the nfs is that much faster than the qs for numbers this big, and since it is asymptotically faster, why would we waste any more time studying the quadratic sieve? We have four reasons.

First, we still question the practicality of the nfs in its present form. Both qs and nfs require solving a very large, sparse matrix before the factorization can be completed. For the qs RSA-129 factorization, the matrix had about half a million rows and columns. Compare that to the nfs RSA-130 factorization where the matrix had about 3.5 million rows and columns! Solving this matrix took 67.5 CPU-hours and about 700 megs of memory on a Cray C-90 supercomputer. Extrapolating from this, factoring a 512-bit number (about 155 digits) with nfs would require solving a matrix with about 19 million rows and columns. This would take a few gigabytes of memory and at least a couple months CPU-time using the same algorithm on the same computer. It is not so easy to get access to a supercomputer like this for a large memory job that will run for several months. Thus, the matrix stage seems to be becoming the bottleneck of the nfs, and may prevent it from being used for much larger factorizations. In contrast, factoring a 512-bit number with qs would require solving a matrix of approximately 5.5 million rows and columns, which is only slightly bigger than that which was solved for RSA-130. These extrapolations are worked out in appendix B.

Second, if the researchers that factored RSA-129 had programs available to them to solve larger matrix problems, then a larger factor base could have been used for RSA-129 which might have significantly increased its speed. According to the asymptotic analyses, the number of rows and columns in the matrix should be approximately the square root of the running times for both nfs and qs. So if factoring a number with qs required k times as much time as that required by nfs, then we would expect the qs matrix to be about k^2 times the size of the nfs matrix. The discrepancy between the matrix sizes and running times for RSA-129 and RSA-130 suggests that

maybe the RSA-129 factor base size was far from optimal. Note, however, that even with their choice of factor base, RSA-129 was factored somewhat efficiently. On the other hand, nobody has yet demonstrated that the nfs is practical with small factor bases.

Third, the RSA-129 factorization could have been done significantly faster by using a version of the qs known as the self-initializing quadratic sieve (siqs). Siqs would have required more computer memory than the qs program used for RSA-129, but not much more than the nfs programs that were used for the RSA-130 factorization. One of our main goals of this thesis is to convince the reader that siqs is at least twice as fast as the version of qs which was used for the RSA-129 factorization.

Fourth, we remark that for those people who wish to factor smaller numbers (say, below 100 digits), qs is unquestionably much faster than nfs, and siqs is the most optimized version of qs.

Chapter 2

Quadratic Sieve

Suppose we want to factor an integer, N. One way of doing this is to find a random relation of the form $X^2 \equiv Y^2 \bmod N$. If $X \not\equiv \pm Y \bmod N$, then the greatest common divisor of $X - Y$ and N will be a proper factor of N. This greatest common divisor can be computed quickly with the Euclidean algorithm.

To find X, Y with $X^2 \equiv Y^2 \bmod N$, we first find several relations of the form $u^2 \equiv v \bmod N$ where v factors into small primes and $u^2 \neq v$. Such relations will be called **smooth**. Let the letter F represent the upper bound for the largest prime that we will allow to divide the v's (we will say more about how F should be chosen later). If more of these relations are obtained than there are primes less than F, then one can use some collection of the smooth relations to construct a relation of the form $X^2 \equiv Y^2 \bmod N$. Finding which collection of relations to use is a linear algebra problem, which we describe in chapter 4.

This is the approach of the quadratic sieve, as well as some other factoring algorithms. But how do we efficiently find the smooth relations? This is where the sieve comes in. We introduce a quadratic polynomial: $g(x) = (x + b)^2 - N$, where b is chosen to be $\lceil \sqrt{N} \rceil$ (the smallest integer which is larger than the square root of N). If, for some integer x, $g(x)$ factors into primes less than F, then taking $u = x + b$ and $v = g(x)$ gives a smooth relation $u^2 \equiv v \bmod N$.

For each prime p less than F, we can determine the values of $g(x)$ that are divisible by p by solving for x in the congruence $(x + b)^2 - N \equiv 0 \bmod p$. If N

does not have a square root mod p, then there are no solutions. This implies p does not divide any values of $g(x)$. Otherwise, let t and $-t$ be the two solutions to $t^2 \equiv N \bmod p$. Solutions to the congruence above are $x \equiv \pm t - b \bmod p$. Notice that if x is one solution, then $x+p$, $x+2p$, $x+3p$, ... are also solutions. Because of this, we can sieve some interval to mark all the places which are divisible by p. After doing this for all $p < F$, we go back to see which values of $g(x)$ are divisible by enough of these small primes to signify a smooth relation. One way of identifying these places is to look about for values of x where the product of the primes is approximately equal to $2x\sqrt{N}$. A more practical way altogether is to sieve with the logarithms of the primes: wherever p divides $g(x)$, add $\log p$ to location x in a sieve array. Then the desired locations are those that have accumulated a value of approximately $\log(2x\sqrt{N})$. Some error must be allowed for rounding, and also because we do not sieve with prime powers.

Since only about half of the primes $p < F$ will have solutions to $t^2 \equiv N \bmod p$, we only have to sieve with these primes. They are known as the **factor base** primes. This also means that when we want to find the relation $X^2 \equiv Y^2 \bmod N$, we only need to get more smooth relations than there are primes in the factor base.

If F is chosen in an optimal way, asymptotically about $e^{(\frac{1}{2}+o(1))\sqrt{\log N \log\log N}}$, then the running time of the quadratic sieve is $e^{(1+o(1))\sqrt{\log N \log\log N}}$. (This is not rigorously proved. It is derived by heuristic arguments. See appendix A.) This is sub-exponential. Notice that the running time depends only on the size of the number N and not the size of its prime factors. The quadratic sieve is summarized in Figure 2.1.

A few years after Pomerance published the algorithm, some researchers programmed it and had immediate success. Before long, an improvement in speed was found which involved replacing $g(x)$ with several different polynomials. This is known as the multiple polynomial quadratic sieve (mpqs).

(qs algorithm to factor N)

Compute startup data: Choose F, the upper bound for factor base primes. Let $b = \lceil \sqrt{N} \rceil$ and $g(x) = (x+b)^2 - N$. Determine the factor base primes: those prime $p < F$ such that there is a solution to $t^2 \equiv N \bmod p$. For each factor base prime p, compute t, a modular square root of N mod p. Then compute and store $soln1_p = t - b \bmod p$, $soln2_p = -t - b \bmod p$, and $l_p = \lfloor \log p \rceil$ (rounding off).

Sieve stage: Initialize a sieve array to 0's. For each odd prime p in the factor base, add l_p to the locations $soln1_p + ip$ and $soln2_p + ip$ of the sieve array, for $i = 0, 1, 2, \ldots$. For the prime $p = 2$, sieve only with $soln1_p$.

Trial division stage: Scan sieve array for locations x that have accumulated a value of at least $\log(2x\sqrt{N})$ minus a small error term. Trial divide $g(x)$. If $g(x)$ factors into primes less than F, then save smooth relation. After scanning entire sieve array, if we have more smooth relations than primes in the factor base, then go to linear algebra stage. Otherwise, continue sieving stage.

Linear algebra stage: Solve linear algebra problem described in chapter 4. For each null space basis vector, construct relation of form $X^2 \equiv Y^2 \bmod N$. Attempt to factor N by computing $\gcd(X - Y, N)$. If all null space vectors fail to give factorization, then return to sieving stage.

Figure 2.1: summary of basic qs algorithm

The problem with using only one polynomial is that the values of $g(x)$ increase in size as x gets bigger. This makes them less likely to be smooth. Thus, as the algorithm progresses, smooth relations become less frequent (see [9]). To escape this problem, we want to be able to change to a new polynomial when the residues of the current polynomial become too big.

In mpqs, Peter Montgomery suggested sieving polynomials of the form

$$g_{a,b}(x) = (ax + b)^2 - N = a^2x^2 + 2abx + b^2 - N$$

where a, b are integers. The graph of $g_{a,b}(x)$ is a parabola, and if we force $0 < b < a$, then it will have axis of symmetry between -1 and 0. We will be sieving on this polynomial for values of x in an interval, say $[-M, +M]$ where M is some integer. The largest values of $g_{a,b}(x)$ for $x \in [-M, +M]$ are $\approx a^2M^2 - N$ and the smallest

$\approx -N$. We take $a \approx \frac{\sqrt{2N}}{M}$ to make the largest and smallest values about the same size in absolute value.

Montgomery suggested choosing b so that $b^2 - N$ is divisible by a, say $b^2 - N = ac$ for some integer c, so that $g_{a,b}(x) = a(ax^2 + 2bx + c)$. This guarantees that every value we sieve on will be divisible by a. Moreover, taking $u = ax + b$ and $v = g_{a,b}(x)$ gives a relation of the form $u^2 \equiv v \bmod N$. Suppose we choose $a = q^2$ for some integer q. If $ax^2 + 2bx + c$ factors into primes $< F$, then the relation of the form $(ax + b)^2 \equiv q^2(ax^2 + 2bx + c) \bmod N$ is just as good as a smooth relation. For example, we may write it as $((ax + b)q^{-1})^2 \equiv ax^2 + 2bx + c \bmod N$. So we are essentially sieving on the $2M + 1$ values of $g_{a,b}(x)/a = ax^2 + 2bx + c$, which are all bounded above by $\approx M\sqrt{N/2}$.

The condition that $b^2 - N$ is divisible by a means that $b^2 \equiv N \bmod q^2$. In particular, if we choose q as a prime such that N is a quadratic residue mod q, then we can quickly compute a value for b using a modular square root algorithm and lifting via Hensel's lemma. The mpqs is summarized in Figure 2.2.

But how big should M be? If M is chosen very large then each polynomial will yield many values to sieve on, but the size of these residues will be rather large. This would not be good, since the entire reason why we are using multiple polynomials is so that the residues are small. So to keep the residues as small as possible, we should choose M very small. Of course we don't get many residues per polynomial, but who cares? We can always generate as many new polynomials as we need so that we have enough to sieve on.

The reasoning above is not quite correct, because it ignores the fact that generating new polynomials takes time. Every time coefficients a, b are chosen for a new polynomial $g_{a,b}(x)$, we must do some work to determine where the factor base primes divide $g_{a,b}(x)$. This is called the initialization stage. For each prime p in the factor base, we again let t denote a solution to $t^2 \equiv N \bmod p$. $g_{a,b}(x)$ is divisible by

p for $x \equiv a^{-1}(\pm t - b) \bmod p$ (assuming p does not divide a, but let's ignore that for the moment). This computation requires doing multi-precision arithmetic (reducing $a \bmod p$ and $b \bmod p$ for all primes in the factor base) and computing an inverse. Now if M is chosen too small, then we will be spending most of our time on the initialization stage and very little time sieving. This is wasteful.

So somewhere there is a nice value of M, not too small nor too large, which optimizes the speed of the algorithm for the number being factored. Silverman gives a table of good parameters (values for F and M) for numbers up to 66 digits [25].

Realize that the reason why we couldn't choose M too small was because we would be wasting too much time on the initialization stage. However, if the initialization stage took no time, then certainly a much smaller M would be better, since this would give smaller residues at no cost, and therefore smooth residues would be found more frequently. It then should not sound too unreasonable to believe that the faster we can perform the intialization, the more we would benefit from using a smaller M.

How much of a speedup can we get with a smaller M? Assuming the polynomial values of $g_{a,b}(x)$ are randomly distributed, we can expect 1 out of every u^u values to be smooth, where $u = \frac{\log M\sqrt{N}}{\log F}$ [4]. Asymptotically, $\log F \approx \frac{1}{2}\sqrt{\log N \log\log N}$ (see appendix A). Let's consider numbers N around 10^{70}. Suppose M is 4 million, which is a reasonable choice determined from our own experimentation. Then $u \approx 6.694$, so approximately 1 in every 336,000 values will be smooth. If we take M one-tenth as big, then $u \approx 6.533$, so about 1 in every 211,000 values will be smooth. Though the u^u approximations are not always very accurate, their ratios usually are fairly good approximations to the true ratio. So, with M one-tenth as big, smooths occur ≈ 1.6 times more frequently. Assuming the ratio of times for the sieving stages and initialization stages are the same in both cases, we expect a speedup factor of 1.6. Taking M one-twentieth as big, then the speedup factor is ≈ 1.8. Note, however,

that in practice, people always use smaller values for F than the asymptotic formula. For our implementation, we found $F = 350{,}000$ to be optimal for 70 digit numbers. Using this value of F, the speedup factors become about 1.7 and 2.0 respectively.

In 1988, several methods for speeding up the initializing stage of mpqs were published in [23]. At the end of section 5 of this paper, the authors comment about one particularly fast way of initializing, but suppress most of the details. This was later published in [1], and is known as the self initializing quadratic sieve (siqs).

2.1 The Self Initializing Quadratic Sieve

The self initializing quadratic sieve provides us with a fast way to change polynomials. This makes it beneficial to use a smaller M than in mpqs. Since M is smaller, the residues are smaller and smooth relations are found more frequently. This is why we expect siqs to be significantly faster than mpqs.

The siqs uses the same polynomials as mpqs, $g_{a,b}(x) = (ax + b)^2 - N$ where $b^2 - N = ac$. Again, we have $g_{a,b}(x) = a(ax^2 + 2bx + c)$. For mpqs, we were able to ignore the a term in front because we took it as a square of a large prime. For siqs, we take a to be a product of primes in the factor base, so that $g_{a,b}(x)$ is smooth if and only if $ax^2 + 2bx + c$ is smooth. Furthermore, this allows it to get several polynomials for each value of a, since there can be several choices for b corresponding to each a. Suppose the leading coefficient is $a = q_1 \cdots q_s$, where the q_l's are distinct odd primes in the factor base. To generate a polynomial, we need to find b satisfying $b^2 \equiv N \bmod a$. There are actually 2^s distinct $b \pmod a$ that satisfy this, but only half of the b's will be used since $g_{a,b}(x)$ gives the same residues as $g_{a,-b}(x)$. Hence we can get 2^{s-1} polynomials.

We want to construct integers B_l $(1 \le l \le s)$ that have the property

$$B_l^2 \equiv N \bmod q_l \text{ and } B_l \equiv 0 \bmod q_j \text{ for } 1 \le j \le s,\ j \ne l.$$

Then the Chinese Remainder Theorem tells us that the square of any combination of the form $\pm B_1 \pm \ldots \pm B_s$ will be congruent to N modulo a (the cross terms of the form $\pm B_i B_j$ with $i \neq j$ disappear since they are divisible by all the q_l's). In other words, these are exactly the values of b. Since we only want one of each b,$-b$ pair, we can fix the sign of B_s to be positive. Now the reader can verify that the B_l are given by the formula $\frac{a}{q_l} \times \gamma_{a,q_l}$ where γ_{a,q_l} is the smallest positive residue of $t_{q_l} \times (a/q_l)^{-1} \bmod q_l$ and $t_{q_l}^2 \equiv N \bmod q_l$. There are two choices for t_{q_l}. We choose the one that makes γ_{a,q_l} smaller.

We take b_1 to be $B_1 + \ldots + B_s \bmod a$. The remaining values of b can be quickly computed by applying a formula for a Gray code:

$$b_{i+1} := b_i + 2(-1)^{\lceil i/2^\nu \rceil} B_\nu$$

where $2^\nu \parallel 2i$ for $i = 1$ to $2^{s-1} - 1$. For information on Gray codes, see appendix C.

Example: Suppose $s = 3$ and $q_1 = 5$, $q_2 = 7$, and $q_3 = 11$ ($a = 385$). We will compute four solutions to $b^2 \equiv 291 \bmod 385$. The reader should verify $B_1 = 154$, $B_2 = 110$, and $B_3 = 70$. So $b_1 := B_1 + B_2 + B_3 = 334$. Then $b_2 := b_1 - 2B_1 = 26$, $b_3 := b_2 - 2B_2 = -194$, and $b_4 := b_3 + 2B_1 = 114$. Notice that each b_{i+1} is obtained from b_i by doing a single addition or subtraction of one of the saved values of $2B_\nu$.

By doing a little precomputation, the initialization stage can be done quickly for the 2^{s-1} polynomials. For the first polynomial, solve $g_{a,b_1}(x) \equiv 0 \bmod p$ for each prime p in the factor base except the primes dividing a.[1] Then compute and store in memory the values

$$2B_j a^{-1} \bmod p \qquad \text{for } j = 1 \text{ to } s-1$$

for all p in the factor base which do not divide a. Now, to initialize polynomial $g_{a,b_{i+1}}(x)$ for the prime p, $g_{a,b_{i+1}}(x) \equiv 0 \bmod p$ must be solved. But this can be done

[1] For primes q_l dividing a, $g_{a,b}(x)/a$ is divisible by q_l if and only if $x \equiv -c(2b)^{-1} \bmod q_l$. However, one may choose not to sieve with these primes. Just allow a little extra error when searching for smooth reports to make up the difference.

quickly by taking the solutions to $g_{a,b_i}(x) \equiv 0 \bmod p$ and adding $(-1)^{\lceil i/2^\nu \rceil}$ times the saved value of $2B_\nu a^{-1} \bmod p$, where $2^\nu \parallel 2i$, and then reducing mod p if necessary. This requires only a few single precision additions, which takes much less time than the multi-precision arithmetic and the computation of inverses required by mpqs. Altogether, setting up the Gray code for the 2^{s-1} polynomials can be done in the same order of time that it would take mpqs to initialize only s polynomials. Thus, the amortized cost of initialization is greatly reduced. We refer the reader to Figure 2.3 for a summary of the siqs initialization stages.

We have two important comments. Although we generally want a to have as many divisors as possible, the q_l dividing a should not be too small. This is because each q_l divides only one out of every q_l sieve locations, and the other primes p divide 2 out of every p sieve locations (since there are 2 square roots mod p). So choosing small q_l can significantly decrease the probability of finding smooth residues. In our implementation, the divisors were always chosen larger than 2,000. Moreover, small prime divisors can also increase the probability of getting redundant relations (essentially duplicate data), which is talked about in the next chapter. The second comment is that siqs requires more memory than mpqs. However, one need not store all the values of $2B_j a^{-1} \bmod p$ if memory is limited. The higher indices are used much less frequently. A speedup over mpqs can still be obtained if one can only store the values of $a^{-1} \bmod p$ and $2B_j a^{-1} \bmod p$ for $j = 1$ and 2, for example. This is explained in [1].

Our goal is to show the practical side of siqs. We have programmed both mpqs and siqs, and have repeatedly experimented with these programs on several numbers of different sizes. We give several tables showing the results of our experiments. These tables suggest that factoring a large number with mpqs will take about twice as much time as factoring that number with siqs, assuming optimal parameters are chosen at least for mpqs.

We want to emphasize that our mpqs and siqs programs were written exactly for comparison reasons. First we programmed mpqs, and then we built onto it the routines necessary for siqs. The two programs have essentially the same sieving routines (there are a few extra comparisons needed in the siqs, but the amount of time for these is negligible). Our sieving routines have not been highly optimized, but any significant improvement to one program can also be applied to the other.

The next four sections describe our comparisons between siqs and mpqs. In section 3.8, details are given about our 116 digit record factorization for the Cunningham project.

(mpqs algorithm to factor N)

Compute startup data: Choose F, the upper bound for factor base primes. Choose M: we sieve each polynomial in the interval $[-M, M]$. Determine the factor base primes: those prime $p < F$ such that there is a solution to $t^2 \equiv N \bmod p$. For each factor base prime p, compute t, a modular square root of N mod p, and store in $tmem_p$. Also store $l_p = \lfloor \log p \rceil$ (rounding off).

Initialization stage: Find a prime $q \approx \sqrt{\frac{\sqrt{2N}}{M}}$ such that N is a quadratic residue mod q. Let $a = q^2$. Compute b, a modular square root of N mod a. $g_{a,b}(x)$ is the polynomial $(ax + b)^2 - N$. For each prime p in the factor base, compute $soln1_p = a^{-1}(tmem_p - b) \bmod p$ and $soln2_p = a^{-1}(-tmem_p - b) \bmod p$.

Sieve stage: Initialize a sieve array of length $2M + 1$ to 0's. Assume the indices of the sieve array are from $-M$ to $+M$. For each odd prime p in the factor base, add l_p to the locations $soln1_p + ip$ for all integers i that satisfy $-M \leq soln1_p + ip \leq M$. Similarly, add l_p to the locations $soln2_p + ip$ for all integers i that satisfy $-M \leq soln2_p + ip \leq M$. For the prime $p = 2$, sieve only with $soln1_p$.

Trial division stage: Scan sieve array for locations x that have accumulated a value of at least $\log(M\sqrt{N})$ minus a small error term. Trial divide $g_{a,b}(x)$. If $g_{a,b}(x)$ factors into primes less than F, then save smooth relation. After scanning entire sieve array, if we have more smooth relations than primes in the factor base, then go to linear algebra stage. Otherwise, go to initialization stage.

Linear algebra stage: Solve linear algebra problem described in chapter 4. For each null space basis vector, construct relation of form $X^2 \equiv Y^2 \bmod N$. Attempt to factor N by computing $\gcd(X - Y, N)$. If all null space vectors fail to give factorization, then return to sieving stage.

Figure 2.2: summary of mpqs algorithm

Initialization stage for first polynomial:
Find primes $q_1, \ldots, q_s$ in the factor base whose product is $\approx \frac{\sqrt{2N}}{M}$. Let $a = \prod_{l=1}^{s} q_l$.
For $l = 1$ to s
    Compute $\gamma = tmem_p \times (a/q_l)^{-1} \bmod q_l$.
    If $\gamma > \frac{q_l}{2}$ then replace γ with $q_l - \gamma$.
    Let $B_l = \frac{a}{q_l} \times \gamma$.
For each prime p in the factor base that does not divide a
    Compute $ainv_p = a^{-1} \bmod p$
    For $j = 1$ to s
        Compute $Bainv2_{j,p} = 2 \times B_j \times ainv_p \bmod p$
Let $b = B_1 + \ldots + B_s$ and $g_{a,b}(x) = (ax + b)^2 - N$.
For each prime p in the factor base that does not divide a
    Compute $soln1_p = ainv \times (tmem_p - b) \bmod p$
    Compute $soln2_p = ainv \times (-tmem_p - b) \bmod p$

Initialization stage for the next $2^{s-1} - 1$ polynomials:
(Assume we just sieved polynomial $\#i$, so we are initializing polynomial $\#(i+1)$, $1 \leq i \leq 2^{s-1} - 1$)
Let ν be the integer satisfying $2^\nu \parallel 2i$.
Let $b = b + 2 \times (-1)^{\lceil i/2^\nu \rceil} \times B_\nu$ and $g_{a,b}(x) = (ax + b)^2 - N$.
For each prime p in the factor base that does not divide a
    Compute $soln1_p = soln1_p + (-1)^{\lceil i/2^\nu \rceil} \times Bainv2_{\nu,p} \bmod p$
    Compute $soln2_p = soln2_p + (-1)^{\lceil i/2^\nu \rceil} \times Bainv2_{\nu,p} \bmod p$

Figure 2.3: summary of siqs initialization stages

Chapter 3

Computational results

In order to say factoring algorithm A is so many times faster than algorithm B, it is necessary to find the best time one can factor a number using algorithm A, and the best time with algorithm B. This should be done with many different numbers. Even though two algorithms may be very similar, it is wrong to assume that the parameters that give the best time for one algorithm will also give the best time for the other.

In addition, if one is to claim that algorithm A is twice as fast as algorithm B, then that person should also say how this was determined. For example, how many numbers were the algorithms tested for? How much were the parameters varied, and which parameters were varied? How much variance was there in the timings? Is there anything left untested that might affect the results? Unfortunately, these commonsense guidelines have not always been followed in the factoring literature.

Our goal is to convince the reader that siqs is about twice as fast as mpqs, at least for the size numbers we are factoring. We give tables of our results so that the reader can draw their own conclusions. Our programs are available by e-mailing the author.[1]

Most of the time in siqs and mpqs will be spent on sieving, and most of the time for sieving will be memory access. In order to compare these algorithms in a fair way, one must access memory as efficiently as possible. In situations like this, it is

[1] e-mail address is scontini@alpha.math.uga.edu

often beneficial to do some *extra* computations so that one can take full advantage of cache memory. Cache memory is fast-access memory that the operating system controls. When an operating system sees a block of memory that is being accessed frequently, it is likely that this memory will be put into cache so that the user can access it quicker.

When sieving, one does not want to put the entire sieve array in memory: sieving on the whole array at once is an inefficient use of cache. A considerable speedup can be obtained by breaking up the sieve interval into blocks. Sieving should be done by all primes for each block before going to the next block. This requires doing a little extra work, but the net speedup can be significant.

It is absolutely necessary that this is done when comparing two algorithms like mpqs and siqs, and also when trying to find optimal parameters. For one thing, siqs will naturally have a smaller optimal value for M than mpqs. If one attempts to sieve the entire array at once, then siqs has the unfair advantage that its sieve array is more likely to fit in cache than the mpqs sieve array. This, for example, could give the false impression that siqs is five times as fast as mpqs, when in fact it may only be twice as fast. As another example, if the entire array is sieved at once, then the limited size of cache memory may cause a person to incorrectly conclude that a small value of M is best, when in fact they could factor much faster by using a larger M and sieving in blocks. Therefore, one should reject any comparison results or parameter results if the sieve interval is not broken into blocks.

The exact amount that cache memory can help depends on the particular computer. The popular benchmarking program "nsieve" provides us with examples. This is available via anonymous ftp at 'ftp.nosc.mil' in directory '/pub/aburto'. The program performs the Sieve of Eratosthenes on an array of length 8191 bytes for a 'High MIPS' rating and on an array of 2,560,000 bytes for a 'Low MIPS' rating. Taking an example from a table on the ftp site, a DEC 3000/500X with operating

system OSF/1 V1.3-3 has High MIPS 105.7 and Low MIPS 29.2. This shows that it is over three times faster on small arrays than it is on large arrays.

Our timings were done on Sun SPARC 5 workstations, each having a 85 Mhz MicroSPARC processor, 64 Kbyte on board cache, and 32 Mbytes of RAM. The nsieve program rated them as having High MIPS 39.0 and Low MIPS 30.6. Since for these computers the High and Low ratings are relatively close, we should not expect to see a great difference in timings when breaking the sieve interval into different block sizes. Nevertheless, we experimented with several different values. This is referred to as the $BLOCKSIZE$ in our tables.

When M is chosen too small or too large for a particular number, the timings will be high. When M is chosen just right, the timing should be nearly minimized. We use the word "nearly" here since sometimes a suboptimal M will give a better timing due to "luck". This can happen because different M correspond to sieving on different polynomials: recall, the leading coefficient is determined by M.

Determination of the optimal factor base depends on whether the double large prime variation [14] or the single large prime variation is used. In the single large prime variation, if a residue is found which has all its primes less than F except one large prime between F and F^2, it is kept. This is called a **partial** relation. If another partial is found with the same large prime, the two can be matched to get a new smooth relation. Finding these partials takes no extra time. However, in practice it approximately doubles the net speed in factoring since usually one needs about half as many "true" smooths as primes in the factor base, and then the remaining smooths can be obtained by matching the partials. If a small factor base is chosen, then partials help more: one will find that they need a smaller fraction of true smooths before the rest of smooths can be obtained from the partials. Similarly, they help less the larger the factor base used for a particular number.

In the double large prime variation, one uses the single large prime variation and also tries to find relations that have two large primes between F and F^2. These are called **partial-partial** relations. Finding the partial-partials does require extra time. After factoring out the factor base primes from a polynomial value, if the remaining part is slightly larger than F^2, then one can attempt to factor it to obtain a partial-partial. However, it is possible that the remaining part is prime, in which case the relation is probably not useful. Therefore, it is usually beneficial to do a quick pseudoprime test before attempting to factor it. The usefulness of the double large prime variation also depends on the choice of the factor base. Many people have reported that factoring large numbers with the double large prime variation is at least twice as fast as factoring with just the single large prime variation.

In practice, one does not accept all partials and partial-partials that have primes between F and F^2. There will be a huge amount of these, and the relations with the largest primes are unlikely to be useful. Therefore, many people choose a cutoff, such as $128 \times F$, for the largest primes that will be accepted.

Our implementation did not include the double large prime variation for a few reasons. The main reason is that it does not seem to help at all for numbers of the size that we are factoring. It also makes it difficult to predict the time to complete a factorization. For example, Lenstra and Manasse observed that the number of smooths obtained from combining partials and partial-partials may differ greatly when factoring two numbers of the same size with the same parameter settings [14]. This makes it very time consuming to find the optimal factor base size. We also do not believe the double large prime variation will have much of an affect on our conclusion, that siqs is about twice as fast as mpqs. One may argue that it causes more time to be spent on the sieving stage (i.e. factoring the partial-partials), and in order to optimize the ratio of sieving time to initialization time, a smaller M should be used. This smaller M might have more of an impact on mpqs than it does

have on siqs. However, for large numbers, any report of a possible useful relation is very rare, so the amount of time on the sieving stage will increase by only a small amount.

We do not show many results on how we determined good factor base sizes for the numbers in our tables. Some of our determination of factor bases was based on experience from previous factorizations, which have not all been recorded. However, we do show two tables that indicate the accuracy of our choices for a 70 and an 80 digit number. We believe the factor bases we have chosen are near optimal for the single large prime variation. If the double large prime variation is used, then it is likely that a smaller factor base is better.

It has been noted [1] that redundant relations are occasionally obtained with siqs. By "redundant relation" we mean either a duplicate relation, or one relation that is easily derived from another. For example, a relation that is a perfect square times another is redundant. We have observed that redundant relations occur most often when the leading coefficients of two polynomials have all except one prime in common. Therefore, we programmed our siqs so that the leading coefficients of any two of our polynomials will differ by at least 2 primes. This greatly reduced the redundancy. For all complete factorizations that we did, the number of redundant relations was less than 1% of the total number of smooths plus number of matches obtained.

Our tables are in appendix D. They present a small portion of experiments we did.[2] We expect that this should be enough to convince the reader of the results that we believe to be true. We have done a lot more experimenting with mpqs parameters than siqs parameters. The lack of effort to determine the best siqs parameters was due to the fact that usually our first or second siqs timing was about twice as fast as the best mpqs timings. Thus, our data almost always suggests siqs is *at*

[2]Contact the author for a complete list of all timings and the numbers we factored.

least twice as fast as mpqs. We also chose F in away that seemed to optimize the mpqs performance. This same value of F may not necessarily optimize the siqs performance. In section 3.5, we explain why the optimal mpqs F is probably too large for siqs. However, choosing the factor base *slightly* too large generally does not make much of a difference in timings, so to be fair, this probably does not change the results very much.

It is not necessary to generate enough data to factor a number to know about how long it will take to get that data (see next section). So our tables contain the amount of time it took to get a fraction of the necessary relations. This fraction should be enough so that one can, with small error, predict the amount of time necessary to completely factor the number. Some complete factorizations were done, however, and these are reported (section 3.1 and section E).

Our timings were done using a timing command provided by the operating system of our workstations. Experienced programmers will realize that such timers are often inconsistent: running the same program twice may give two very different timings, even though they did the exact same operations. We were fortunate that our timer was quite consistent. Most of the repeated timings that we did differed by less than 2% of each other. Our tables contain the average time per experiment when multiple timings were performed.

For parameter choices which were clearly far from optimal, we have inserted 'n.a.' in the tables which means that the data is not available. The 'f.b.' is an abbreviation for factor base.

3.1 60 DIGIT NUMBERS

Our first 60 digit number is a cofactor of $3^{131} + 1$, and the next two were generated as a product of two randomly chosen 30 digit primes. All siqs timings for 60 digit

numbers had 7 primes dividing the leading coefficients, selected anywhere from the 170th prime in the factor base to the 380th. We used $F = 60{,}000$ for the upper bound for factor base primes. This value was determined from much experimentation in the past, and it is consistent with Silverman's choice [25].

Table D.1 (in appendix D) shows our experiments with different values for $BLOCKSIZE$ for our first 60 digit number. We fixed $M = 600{,}000$ and ran our mpqs program until we had 100 smooth relations. The best timings occurred with $BLOCKSIZE$ values in the range of 100,000 to 200,000.

Table D.2 summarizes the amount of time our siqs and mpqs programs spent to get at least 100 smooths for the first 60 digit number using $BLOCKSIZE = 100{,}000$. For this number, the factor base had 3,101 primes, so the amount of smooths we are getting is about 3.2% of the number of primes in the factor base. A higher number of smooths would have given us a better view of which parameters are best. However, this is sufficient enough to illustrate at least one point.

Notice that the best mpqs time of 2 minutes and 26 seconds occurred when $M = 600{,}000$, where it obtained an average of $\approx .685$ smooths per second. For this M, the siqs time is 1 minute and 52 seconds, only slightly better than the mpqs timing. But the difference is that using a smaller M makes the mpqs timing worse and the siqs timing better. The mpqs timing with $M = 100{,}000$ is more than twice as bad as the mpqs timing with $M = 600{,}000$. This is because when a value of M is chosen too small, mpqs will be spending a large percentage of its time on the initialization stage, and a small percentage of time actually searching for the good data. Since siqs performs the initialization so fast on average, we are able to get a much better time using a smaller M. The best siqs timing occured when $M = 100{,}000$, averaging ≈ 1.507 smooths per second. Dividing this rate by the best mpqs rate of .685 smooths per second, we see that the siqs performance was about 2.20 times the speed of the mpqs performance.

From the small amount of data in Table D.2 we can predict how much time it will take to get enough data to factor the number. It would be a good guess to say that we need about 1550 smooths (half the number of factor base primes) and then the remaining smooths can be obtained from the partial relations. However, to be absolutely sure we succeed, we decided to go a little beyond this, to 1700 smooths. At a rate of 1.507 smooths per second for the siqs timing with M = 100,000, the expected amount of time is about 1700/1.507 seconds $\approx$ 18.80 minutes. Table D.3 shows the actual time to get at least 1700 smooths with these paramters was 20 minutes and 45 seconds. The error of about 10% could have been reduced by getting more smooths in our original experiment. The prediction for mpqs with M = 600,000 is 1700/.685 seconds $\approx$ 41.36 minutes. The actual time was 40 minutes and 33 seconds.

We mentioned in the previous section that the siqs algorithm can generate a small amount of redundant data. For the last siqs timing in Table D.3, we had 1709 smooths and 19868 partials (accepting partials up to size $128 \times F$). Only one of the smooths was redundant. The partials combined for 2093 matches, of which 26 were redundant. So we had 3775 true complete relations, which was nearly 700 more than we needed.

For our second 60 digit number, we did another cache experiment and again arrived at the conclusion that the optimal $BLOCKSIZE$ is somewhere between 100,000 and 200,000. For brevity, we have omitted this table. Table D.4 contains the timings to get at least 100 smooths for the second 60 digit number, this time using $BLOCKSIZE$ = 150,000. The best mpqs timings occurred with M = 1,200,000 and M = 900,000 both at a rate of $\approx$ .6 smooths per second. The best siqs timing was with M = 150,000 at a rate of $\approx$ 1.085 smooths per second. The ratio of times shows that siqs was about 1.80 times faster than mpqs. Perhaps a better siqs rate could have been accomplished with a smaller M.

The most emphasis should be put on D.5, which contains the timings for the third 60 digit number. Here we got at least 300 smooths for all experiments. Table D.5 contains a column for $BLOCKSIZE$, which we did not keep constant for all timings. With M = 1,200,000 we achieved the best mpqs rate of $\approx$ .476 smooths per second. Compare that to the rate of .997 smooths per second for siqs with M = 100,000. This is 2.09 times faster than the mpqs timing. Table D.5 also illustrates how the times can blow up if an M is chosen too large or too small with mpqs.

For our timings, we found the optimal values of M in our implementations to be about 100,000 (or perhaps smaller) for siqs and somewhere between 600,000 and 1,200,000 for mpqs.

3.2 70 DIGIT NUMBERS

All three of our 70 digit numbers were a product of two thirty five digit primes. In all experiments with siqs, the leading coefficients had 8 prime divisors. These primes were selected anywhere from the 180th to the 380th prime in the factor base.

Our first concern in factoring 70 digit numbers was finding a good factor base size for mpqs. Let us assume for the moment that we are not using the large prime variation. Then one could proceed by first choosing an initial guess for a value of F, then finding out what the best M is for that factor base size, and finally experimentally determine the rate that smooths are being obtained to make an accurate prediction on the expected amount of time to factor the number. Upon doing this for several values of F, one can empirically determine which is best. This is what we did in Table D.6 for our first 70 digit number.

Now, how do large primes affect our results? We mentioned earlier that the large prime variation is less useful for large factor base sizes and more useful for small ones. We have looked at a few examples for 60 and 70 digit numbers to determine just how

much the large primes help. In our examples, we kept the cutoff for large primes at some constant times F. In all cases, a good value of F required approximately half as many smooths as primes in the factor base and then the remaining residues were obtained from the partial relations. Then, upon doubling the value of F and attempting to refactor the same number with the larger factor base, it required about an extra 5% higher ratio of smooths to factor base primes before the rest of the smooths were obtained from the partials. For example, suppose that we chose a good value for F and for it, we needed a number of smooths equal to .47 times the size of the factor base before we had enough relations to factor the number. Then, if we re-ran the program using a value of F twice as big, it would be be necessary to get about .52 times the size of the new factor base smooths before we could factor the number.

With this in mind, we can make a decent guess at how much the large primes help us for various factor base sizes. In our guesses, we always took the smaller factor base when timings were close. From Table D.6, we see that the best time occurred with F = 500,000, but that was not much better than the time with F = 350,000. Based on this and some experiments with other 70 digit numbers, we chose F = 350,000 for the 70 digit number tables in this paper. All tables contain a column for $BLOCKSIZE$.

Table D.7 contains the timings to get at least 1,200 smooths for the first 70 digit number. The best mpqs timing occurs when M = 600,000 and $BLOCKSIZE$ = 200,000. Notice that this is better than using the same value of M but with $BLOCKSIZE$ = 150,000 or $BLOCKSIZE$ = 300,000. The point is that we had to increase the $BLOCKSIZE$ to optimize our factoring when we went to larger numbers. Comparing this mpqs timing to the siqs timing with M = 200,000, we see that siqs was about 2.15 times as fast. Since we did only two siqs experiments it may be even faster than this.

In Table D.8 for the second 70 digit number, the siqs timing with M = 200,000 was 2.16 times faster than the best mpqs timing, with M = 3,600,000.

In Table D.9 for the third 70 digit number, we ran our programs until we got $\geq$ 1,500 smooths. The best siqs timing was 1.97 times faster than the best from mpqs. It is interesting to compare these timings with those from Table D.8. Although the two numbers are the same size in digits, we got smooths for the third 70 digit at a rate of nearly four times as fast as we did for the second. This is mainly due to differences in the factor base. In section 3.7 we give an explanation for this. We probably could have factored the second 70 digit number much faster if we had used a multiplier [21].

Our results indicate that siqs is about twice as fast as mpqs for 70 digit numbers. For mpqs, our optimal M seems to be in the range of 3,000,000 to 6,000,000, and for siqs it seems to be about 200,000.

3.3 80 DIGIT NUMBERS

All three of our 80 digit numbers were generated as a product of two 40 digit primes. In all experiments with siqs, the leading coefficient had 10 prime divisors, selected anywhere from the 170th to the 370th prime in the factor base. Table D.10 shows our experiments with factor base sizes for this first 80 digit number. Based on these timings and some timings from other 80 digit numbers, we chose F = 900,000 in all three tables.

We leave it to the reader to verify that siqs was 2.11, 2.14, and 2.21 times faster than mpqs for the first, second, and third numbers respectively (see Tables D.11, D.12, and D.13).

The careful observer might notice that in these tables, the value of F is larger than twice the value of M for the best siqs timings. This implies that some primes may

not land in the sieve array at all. The natural question to ask is why bother sieving with those primes if it takes time to deal with them and often it is to no benefit at all? Let us give a philosophical answer without going into any mathematical details. The important thing is to consider the amortized cost of operations. A prime may miss some sieve intervals completely, but it will also hit many sieve intervals, and for these it will contribute a lot to the sieve array. This is not an entirely new concept to experienced programmers. Even in mpqs we see examples of this: when the sieve interval is split into blocks to optimize the use of cache memory, not all primes are going to land in every block. But it is necessary to do the extra work of sieving with these primes to optimize the speed of the program.

Our results suggest that siqs is about twice as fast as mpqs for 80 digit numbers. For mpqs, we got our best timings with M between 6,000,000 and 9,000,000. For siqs, $M = 300{,}000$ seems to be a good choice.

3.4 Importance of initialization

Although we suggested values for M for our mpqs program, one should realize that the optimal value is strongly dependent upon implementation. Faster initialization means faster factoring because it allows using a smaller M efficiently.

When we showed the timings above to Arjen Lenstra, who probably has more experience with mpqs than anyone else in the world, he questioned the efficiency of our mpqs initialization stage routines. We gave him a copy of our program, and he timed our initialization against his, and found his to be significantly faster. He then generously gave us a copy of his mpqs program so we could improve on our own. His mpqs initialization used several nontrivial tricks, taking advantage of the fact that the primes in the factor base are less than 26 bits for the numbers we are factoring. We then incorporated his initialization routines into our mpqs program.

After redoing several timings, we found the best mpqs timings with his initialization routines were 5 to 10% better than the best mpqs timings with our original program. This only changes our "factor of 2 speedup" in a minor way. Note too that some of his tricks could also be applied to our siqs program.

It is interesting to compare the amount of time our (original) mpqs and siqs programs spend initializing polynomials. Referring to the third 70 digit number in Table D.9 with M = 3,200,000, a total of 9 minutes was spent on initialization. Of the 9 minutes, 50 seconds was spent on generating the coefficients of the polynomials alone. The total number of polynomials generated was 568. Suppose we wanted to use the same M to factor that number with siqs and let us ask how much time it would take to do the initialization.[3] Since our siqs program selects 8 prime divisors for the leading coefficient, it generates 2^7 polynomials at a time. Therefore only 5 choices for the leading coefficient are needed for siqs to generate at least the same number of polynomials. The amount of time to generate these coefficients and to do the preprocessing stages is only 14 seconds!

3.5 Remarks on factor base sizes

The optimal mpqs F is always too large for siqs. We illustrate this by example for numbers $N \approx 10^{70}$. With mpqs we found the optimal parameters to be $F \approx$ 350,000 and $M \approx$ 4,000,000. For siqs we used $M \approx$ 200,000 and remarked that the factor base size of 350,000 was actually too large.

The polynomials that we are sieving on have residues of size $M\sqrt{N}$ after a is factored out. For mpqs this is $\approx 4 \times 10^{41}$ and for siqs it is $\approx 2 \times 10^{40}$. Since the choice of optimal factor base depends completely on the size of the residues, we can extrapolate to approximate the optimal factor base size for siqs. First note that

[3] Of course, in practice we would use a much smaller M, but the point of this comparison is to show how much faster siqs initializes.

factoring a number equal to about 2.5×10^{67} using $M = 4{,}000{,}000$ with mpqs will yield the same size residues as those that siqs obtains on 70 digit numbers. Thus, the best factor base size for factoring a 70 digit number with siqs is about the same as the best factor base size for factoring numbers near 2.5×10^{67} with mpqs.

Asymptotically, the optimal choice for the upper bound for factor base primes is $F = e^{(\frac{1}{2}+o(1))\sqrt{\log N \log\log N}}$. Ignoring the $o(1)$, we can approximate the ratio of choices for F with $N = 2.5 \times 10^{67}$ to F with $N = 10^{70}$. This turns out to be about 0.725, implying that the optimal siqs F for factoring 70 digit numbers is $0.725 \times 350{,}000 \approx 250{,}000$.

In the above argument we assumed that the best mpqs M for $N = 2.5 \times 10^{67}$ is about the same as the M for $N = 10^{70}$. There may be a slight error in this assumption, since M should decrease as N does. If we assume the best M decreases to as little as 2,000,000, then the calculations tell us that the best siqs F is $\approx 275{,}000$. Therefore it is reasonable to assume F should be somewhere between 250,000 and 275,000. Again, we repeat that choosing F slightly too large usually does not make a big difference in timings. An example is given in the next section.

3.6 Some complete factorizations

For the 70 and 80 digit numbers, we also generated data to completely factor the numbers. Tables E.1 through E.6 in appendix E summarize our results.

For each number we would sieve until we had about half as many smooths as primes in the factor base. We would then check to see if that was enough data to factor the number, and if it was not, we would continue sieving a little more until we could factor the number. Some of the numbers could have been factored a little sooner than the times listed because of the high numbers of matches obtained from partials.

In all of these timings, the number of partials and matches for mpqs was larger than that for siqs. Part of the reason for this is because of implementation: since the residues are all size $M\sqrt{N}$, it is natural to set the threshold for detecting smooth or partial relations to be some function of $M\sqrt{N}$. Since the mpqs M is larger than the siqs M, we will detect more partials when using mpqs. Also, there are not as many matches with siqs since the factor base size we were using was not optimal for siqs. Using a smaller F would have balanced everything out better. Take for example the third 70 digit number with siqs (Table E.3). We initially tried to factor the number after we had 7,500 smooths (almost half the factor base size), but were unsuccessful because we had only 6,699 matches among the partials, and the combination of the 14199 relations was not enough to find a dependency in the matrix. We then sieved until we had at least an extra 200 smooths, which was enough to factor the number. We later tried factoring this using $F = 275{,}000$. The number of primes in the factor base was 12,172, so we sieved until we had 6,014 smooths (a little less than half the factor base size). When sieving was completed, the number of partials was 68,009 which combined to 6,168 matches, allowing us to factor the number. The time for this was 111 minutes and 31 seconds. Notice that with the smaller factor base size, we had more matches than smooths.

3.7 Differences in factor bases

Tables E.2 and E.3 are examples on how two number of the same size can differ by a factor of 3 or 4 in the amount of time necessary to factor them with the quadratic sieve. It is clear that the difference in timings comes from the factor bases. The second 70 digit number only has 6 primes less than 100 in its factor base: 2, 19, 43, 47, 61, and 83. The third 70 digit number has 13 primes less than 100: 2, 3, 5, 7, 19, 23, 31, 37, 43, 47, 61, 71, and 83.

It would be nice to have an estimate of the differences in timings which depends upon the factor base. We suggest using

$$e^{\sqrt{\log N' \log\log N'}} \tag{3.1}$$

where

$$N' = \left(e^{\log\sqrt{N} + \log F - \sum_{p \in \text{f. b.}} E(p)} \right)^2 \tag{3.2}$$

and $E(p)$ is the expected contribution of prime p to a location of the sieve array. That is,

$$E(p) = \begin{cases} (2\log p)/(p-1) & \text{if } p \text{ odd prime} \\ \frac{1}{2}\log 2 & \text{if } p = 2 \text{ and } N \equiv 3 \bmod 4 \\ \log 2 & \text{if } p = 2 \text{ and } N \equiv 5 \bmod 8 \\ 2\log 2 & \text{if } p = 2 \text{ and } N \equiv 1 \bmod 8 \end{cases}$$

The idea behind formulas (3.1) and (3.2) is that when the factor base is rich in small primes, it is like sieving on smaller residues since, after sieving, the expected sizes of the remaining residues are smaller than those from an "average" factor base. We expect that on average, the $\sum_{p \in \text{f. b.}} E(p)$ in (3.2) will be about $\log F$. So the difference $\log F - \sum_{p \in \text{f. b.}} E(p)$ represents the expected amount that the factor base primes contribute below or above the average. The $\log\sqrt{N}$ is the size of the residues that we are sieving on. So the entire exponent, $\log\sqrt{N} + \log F - \sum_{p \in \text{f. b.}} E(p)$, should be thought of as an adjusted residue size, and N' as an adjusted value of N. Equation (3.1) measures the expected time for the adjusted N (ignoring the $o(1)$).

The way to use this formula is to compute the values for two different N of about the same size, and then compute the ratios of the values to get the expected ratios of running times. For the second 70 digit number, the $\sum_{p \in \text{f. b.}} E(p)$ is ≈ 9.4, and for the third, ≈ 13.9. The values of (3.1) are 4.71×10^{12} and 1.83×10^{12} respectively. The ratio is 2.57, meaning that we expect the amount of time to factor the second to be about 2.57 times the amount of time to factor the third. We have two possible

explanations for difference between what our formula predicts and what was actually observed. First note that in Table E.2, we could have stopped sieving earlier, due to the large number of matches obtained. Second, we used a slightly larger value of M for the second 70 digit number.

Equation (3.2) can be written more simply as

$$N' = Ne^{2(\log F - \sum_{p \in \text{f.b.}} E(p))}.$$

We call the value of $e^{2(\log F - \sum_{p \in \text{f.b.}} E(p))}$ the **adjustment factor**. Its value is not very sensitive to the choice for F. This means, for computational purposes one need not compute the entire $\sum_{p \in \text{f.b.}} E(p)$, but instead it will generally be sufficient to sum up to some fixed bound, like 10,000.

3.8 The factorization of a 116 digit number

We have used siqs to be the first to factor about a dozen numbers for the Cunningham project [3]. The largest was a 116 digit divisor of $2^{481} + 2^{241} + 1$, completed on June 15, 1996. This number was on the "Most Wanted List" for the project.

The number we factored was (on the following two lines)

95354559032083752574945877446569580720031903863975472825 45\

42058450367020884041244479727460888919678763367415503868 37

which is the product of the 36 digit prime

258190389365279446113985999067417377

and an 81 digit prime. This is the record for the largest quadratic sieve factorization for the Cunningham project. For general purpose factoring algorithms like qs, the record is measured by the length of the number factored.

There have been three other 116 digit quadratic sieve factorizations for the Cunningham project, but ours has the highest most significant digit. Due to the poor factor base, we had to use a multiplier of 29, which means that we were actually factoring a 118 digit number. Compared to other factorizations, we had a small amount of computing power. We also did *not* use the double large prime variation.

The factorization took approximately 11 workstations and 10 months real time of sieving, although a couple of these workstations were available for only a small fraction of the total sieving time. Eight of the workstations were Sun SPARC 5's (described earlier). The other three were a SPARC 2, a SPARCstation 1000, and a SPARC 20. However, these three had other heavy jobs to run (they were being used as servers), which prevented them from performing better than most of the SPARC 5's. The SPARC 5's were also being used by other graduate students and professors for their research. The most productive computers got an average of about 300 smooth relations per week. For the factor base size of 196,701 primes, we needed about 90,000 smooths and the rest of the smooths came from partial relations.

Based on the figure of 300 smooths per week and the High MIPS rating of 39.0, the total cpu time spent was 224 MIPS years. This is a pessimistic rating for three reasons: it uses the highest MIPS rating of the computer, we actually factored a 118 digit number, and we did not use the double large prime variation.

We can extrapolate from this to determine about how much CPU time is necessary to factor a 129 digit number, and to compare that with the RSA-129 digit number, which took about 5,000 MIPS years. The ratio of times to factor a 129 digit number to factoring a 116 digit number is approximately $e^{\sqrt{\log 10^{129} \log\log 10^{129}}} / e^{\sqrt{\log 10^{116} \log\log 10^{116}}} \approx 12$. So we could factor a 129 digit number in about $12 \times 224 \approx 2{,}700$ MIPS years. And, if we used the double large prime variation (which was used in RSA-129), then perhaps this reduces down to about 1,350 MIPS years.

Comparisons like the one just made are not always trustworthy since, as we have seen (section 3.2), factoring a number with any version of qs may take 3 or 4 times longer than another number of the same size. However, bringing in the adjustment factor (see previous section) may give us a better idea on how accurate our comparison is. Without the multiplier, the adjustment factor for our 116 digit number is ≈ 38.5, which is larger than the RSA-129 adjustment factor of ≈ 7.04. The N' in (3.2) for our 116 digit number is 118 digits, while the N' for RSA-129 is still 129 digits. We can interpret this to mean that without multipliers, our 116 digit number is about as difficult to factor as a typical 118 digit number, while RSA-129 remains as difficult as a typical 129 digit number. With the multiplier of 29, the new adjustment factor for our 116 digit number is $\approx .056$ and the N' is 117 digits. The RSA-129 multiplier was 5, which makes its new adjustment factor $\approx .287$ and its new N' is still 129 digits. Thus, the CPU time comparison above seems to be pessimistic towards our siqs implementation.

We hope this is a convincing argument on the benefits of siqs over the regular mpqs. On the other hand, however, we should mention that some of the computers in the RSA-129 factorization had very small amounts of memory, and because of this, they had to run a suboptimal version of the mpqs code. So perhaps the "5000 mips years" is pessimistic towards the RSA-129 factorization. We also should draw attention to the comments about how processor speed doesn't always correlate well with memory access speed, and therefore MIPS years is not the ideal rating for sieving algorithms (explained in [2]).

Chapter 4

The matrix problem

This chapter describes the matrix problem that must be solved in the qs algorithm, as well as to show past and possible future methods in solving it. As explained in the introduction and chapter 2, this is a very important part of qs. A similar matrix problem must be solved for nfs.

Let's start with an example. The following are smooth relations for the number N=14,137.

$$\begin{aligned}
119^2 &\equiv 24 = 2^3 \times 3 && \bmod N \\
121^2 &\equiv 504 = 2^3 \times 3^2 \times 7 && \bmod N \\
131^2 &\equiv 3024 = 2^4 \times 3^3 \times 7 && \bmod N \\
149^2 &\equiv 8064 = 2^7 \times 3^2 \times 7 && \bmod N \\
151^2 &\equiv 8664 = 2^3 \times 3 \times 19^2 && \bmod N
\end{aligned}$$

The second column of numbers contains the least positive residue of the first column mod N, and the third column contains the factorization of the second column. If we multiply the left hand sides of the first, third, and fourth relations together, that will be congruent to the product of the right hand sides. Writing it in terms of factorization, we have $119^2 \times 131^2 \times 149^2 \equiv 2^{14} \times 3^6 \times 7^2 \bmod N$, which is the same as $(119 \times 131 \times 149)^2 \equiv (2^7 \times 3^3 \times 7)^2 \bmod N$. Thus, we found a relation of the form $X^2 \equiv Y^2 \bmod N$.

Multiplying relations together corresponds to adding exponents of the prime factors, and searching for a perfect square amounts to finding where all exponents sum

to even numbers. So we only care about the exponents in the prime factorizations, and in particular whether they are even or odd. This suggests creating, for each relation, an exponent vector that has length equal to the number of primes in the factor base. The jth entry in that vector represents the exponent reduced mod2 to which the jth prime in the factor base divides the right hand side of that relation. The 5 exponent vectors for the relations above are

primes	v_1	v_2	v_3	v_4	v_5
2	1	1	0	1	1
3	1	0	1	0	1
7	0	1	1	1	0
19	0	0	0	0	0

Now the condition that the product of the first, third, and fourth relations is a perfect square is the same as saying $v_1 + v_3 + v_4$ contains all zero entries mod2. Equivalently, if we combine these five column vectors into a matrix B, then B times the vector $(\ 1\ \ 0\ \ 1\ \ 1\ \ 0\ )^{\mathrm{T}}$ is the zero vector, working over $GF(2)$. Hence, the problem of finding a perfect square is reduced to finding a (nontrivial) null space vector in a matrix. Each null space vector should give us at least a 50% chance of success in factoring, so we can be fairly sure to succeed if we find about 10 independent null space vectors. Notice that there are other null space vectors in the matrix above, such as $(\ 0\ \ 1\ \ 0\ \ 1\ \ 0\ )^{\mathrm{T}}$.

Each relation will have few prime factors, so each column vector of B will have very few nonzero entries, implying that B is very sparse. However, the top rows of B, corresponding to the smallest primes in the factor base, are somewhat dense. Most of the nonzero entries are at the top of the matrix. By the **density** of the matrix we will mean the average number of nonzero entries per column.

4.1 Historical solutions

When qs was first published, it was thought that the matrix stage would dominate the running time. In order to equate the asymptotic running times of the sieving and the matrix stages, it was necessary to choose a suboptimal F [19]. However, researchers eventually found ways to adapt faster matrix methods to the factoring problem, which allowed the use of the optimal F for the sieving stage. With this F, solving the matrix should take about take about as much time (asymptotically) as sieving (see appendix A). However this has never happened. In practice, the time spent on the matrix stage has always been a very small fraction of the sieving time, even when direct methods have been used. So why is the matrix stage a problem? The answer is because of the limited memory available on a single computer, and the lack of clever ways of parallelizing the methods. In this section we outline the past and present methods for solving the matrix.

Initially, Gaussian elimination was used. For an n by n matrix,[1] this takes time $\theta(n^3)$. Gaussian elimination can be implemented very efficiently when working over $GF(2)$. Each entry in the matrix requires only one bit to store, so 32 entries can be stored in one computer word, assuming the computer has 32 bit words. Moreover, adding elements over $GF(2)$ is the same as doing exclusive or's (XOR's) of the bits. Since computers can perform XOR's between two words in the same amount of time they can between two bits, a factor of 32 speedup can be obtained over a naive implementation. Unfortunately, the memory requirements are too much. A matrix with $6 \cdot 10^4$ rows and columns would require $6 \cdot 10^4 \times 6 \cdot 10^4/8$ bytes of memory for storage, which is 450 megabytes. This is a relatively large amount of memory to store a small factoring matrix.

[1] Factoring matrices are usually not square, but very close. We will refer to n by n matrices throughout this section.

A better method, known as structured Gaussian elimination, was presented in [18] [22]. Structured Gauss takes the large sparse matrix and reduces it to a smaller, dense matrix in an intelligent way. The smaller matrix, typically about one fourth or one third the size of the original matrix [11] [12], can then be processed by regular Gaussian elimination. Arjen Lenstra has used this method quite a bit, and has programmed it on a full size MasPar MP-1 massively parallel computer [12]. This implementation was used to solve the RSA-129 matrix [2]. The 524,339$\times$569,466 matrix of density ≈ 47 was reduced to a 188,614$\times$188,160 dense matrix which required over 4 gigabytes to store on disk. The dense matrix was solved in 45 hours.

Clearly structured Gauss requires too much disk space and would not be very fast for solving matrices with over a million rows and columns. So attention has turned to iterative methods. Iterative methods create a sequence of vectors by multiplying the matrix by previous vectors in the sequence. After a certain number of steps, some linear combination of the vectors can be used to solve the system of equations. For an n by n matrix of density d, typical running times are $\theta(dn^2)$. In our case d is very small, so the running times are essentially $\theta(n^2)$. In addition, the memory required to store the matrix is $\theta(dn)$ since it is only necessary to store the indices of the nonzero entries. This is much less than Gaussian elimination.

In [18], Odlyzko showed how to adapt the conjugate gradient method to finite fields. Around the same time, Douglas Wiedemann invented an iterative method that was specifically designed for solving systems of equations over finite fields [26]. Wiedemann's method was based on finding the minimal polynomial of B using the Berlekamp-Massey algorithm [16]. His algorithm required about $3n$ multiplies of B by vectors. Although it was quite exciting, nobody had programmed it and used it for large factoring matrices. So it was not clear that it would beat structured Gauss in practice.

Later, LaMacchia and Odlyzko experimented with the Lanczos method [11]. Lanczos was designed to work over the field of real numbers. They showed how to apply the same formulas over finite fields with a high probability of success. We will comment on their ideas in the next two sections.

Although these methods beat structured Gauss asymptotically, they do not get the factor of 32 speedup that Gaussian elimination obtains by packing 32 field elements into a word when working over $GF(2)$. But why should we not be able to get this speedup with the iterative methods also? Rather than multiplying the matrix by one vector at a time, we could try to multiply it by 32 vectors at a time by storing 32 vectors into an array of words. The jth word would contain the jth entries from each of the 32 vectors. Such a vector is called a block vector, and multiplying a matrix by it can be done in the same time as multiplying a matrix by a single vector. So the question is, can the iterative methods be generalized to work with block vectors rather than single vectors so that 32 vectors can be processed at one time? This would give these algorithms the factor of 32 speedup similar to what we got with Gaussian elimination.

Don Coppersmith showed how to do this first with his block Lanczos algorithm [6]. However, his ideas are somewhat complicated and hard to program. He next developed a block Wiedemann algorithm which is not so difficult to program [7]. Later, Peter Montgomery developed his own block Lanczos algorithm [17] and programmed it to solve some nfs matrices. One of these had 1,284,719 rows and 1,294,861 columns. The fact that he was able to solve a matrix this big not only put a spotlight on his algorithm, but also was one of the final links in making the nfs practical.

For factoring matrices, both block Lanczos methods require about $2n/32$ matrix-block vector multiplies, while the block Wiedemann does $3n/32$. It is possible to do the block Wiedemann with less multiplies but then other parts of the algorithm

become more expensive. Montgomery's block Lanczos requires less block vector inner products and less multiplies of block vectors by 32×32 matrix blocks than Coppersmith's, so it should be faster. In [5], Coppersmith's block Wiedemann was compared to Montgomery's block Lanczos, and in all timings Montgomery's algorithm was faster. They also programmed these algorithms on the MasPar MP-1 massively parallel computer. The parallel block Lanczos was used to solve a 1,472,607 by 1,475,898 nfs matrix in 2.5 CPU days. This was part of a record nfs factorization, a 119 digit partition number (reported in [10]). It was beat two years later by the factorization of RSA-130 [8], which is the record at the time of writing. The ≈ 3.5 million rows and columns RSA-130 matrix was also solved by block Lanczos, but on a Cray C90 supercomputer. Today, most large factorization matrices are being solved by Montgomery's block Lanczos method.

As the matrices get larger, it is becoming more difficult to find a single machine to handle them. It will eventually become necessary to use networks of workstations to work together on solving a single system of equations. In the next two sections we describe Lanczos and ways to apply Lanczos to factoring matrices. We then give a simple method of parallelizing the algorithm on a network of workstations. The main purpose of our method was to test the feasibility of distributing the computation for this algorithm. Our method was somewhat successful, but improvements need to be made to handle large matrices more efficiently. Some improvements and other ideas for parallelizing iterative methods are discussed in the final section.

4.2 Lanczos method

Suppose A is a symmetric, positive definite $n \times n$ matrix with real number entries. To solve $Ax = b$ for $b \in \mathrm{R}^n$, standard Lanczos generates a sequence of vectors $w_0, w_1, \ldots$ by

$$w_0 = b$$

$$w_i = Aw_{i-1} - \sum_{j=0}^{i-1} c_{i,j} w_j \qquad \text{for} \qquad i > 0$$

where the real numbers $c_{i,j}$ are chosen so that the vectors are orthogonal with respect to A: $w_j^{\mathrm{T}} A w_i = 0$ for $i \neq j$. We can determine the $c_{i,j}$ by an inductive argument. Assume by induction that $w_j^{\mathrm{T}} A w_k = 0$ for $j, k < i$, $j \neq k$. Then $w_j^{\mathrm{T}} A w_i = w_j^{\mathrm{T}} A^2 w_{i-1} - c_{i,j} w_j^{\mathrm{T}} A w_j$. We want this to be 0, which is accomplished by taking

$$c_{i,j} = \frac{w_j^{\mathrm{T}} A^2 w_{i-1}}{w_j^{\mathrm{T}} A w_j}.$$

There is no division by 0 here because of the positive definite condition. By the symmetry of A, we also have $w_i^{\mathrm{T}} A w_j = 0$.

Eventually one of the w_i will be zero. To see this, note that $n+1$ vectors in R^n must be linearly dependent. So after n iterations, there will be real numbers $a_0, a_1, \ldots, a_n$ (not all zero) such that $\sum_{j=0}^{n} a_j w_j = 0$. Let a_m be the last nonzero number (i.e. a_m is not zero, but a_j is zero for $j > m$). Then $0 = w_m^{\mathrm{T}} A \sum_{j=0}^{n} a_j w_j = \sum_{j=0}^{n} a_j w_m^{\mathrm{T}} A w_j = a_m w_m^{\mathrm{T}} A w_m$ by the orthogonality condition. By the positive definiteness of A, $w_m^{\mathrm{T}} A w_m > 0$ if $w_m \neq 0$. The previous two sentences imply that w_m must be equal to zero, as we asserted.

The iteration is complete when the zero vector is found. Let

$$x = \sum_{j=0}^{m-1} \frac{w_j^{\mathrm{T}} b}{w_j^{\mathrm{T}} A w_j} w_j.$$

By construction, $w_k^{\mathrm{T}} A x = w_k^{\mathrm{T}} b$ for $0 \leq k \leq m-1$, and so $w_k^{\mathrm{T}}(Ax - b) = 0$. Equivalently, if we multiply the transpose of any vector in $< w_0, \ldots, w_{m-1} >$ by $Ax - b$, we get 0. We claim $Ax = b$. Note that $Aw_k = w_{k+1} + \sum_{j=0}^{k} c_{k+1,j} w_j$, so each Aw_k is in $< w_0, \ldots, w_{m-1} >$. Since $b = w_0$ and Ax is a linear combination of the Aw_k's, we know that $Ax - b$ is also in $< w_0, \ldots, w_{m-1} >$. Thus, there are

real numbers $d_0, \ldots, d_{m-1}$ such that $Ax - b = \sum_{j=0}^{m-1} d_j w_j$. Multiplying through by $w_k^{\mathrm{T}} A$, we get $w_k^{\mathrm{T}} A(Ax - b) = d_k w_k^{\mathrm{T}} A w_k$. On the other hand, $w_k^{\mathrm{T}} A = (Aw_k)^{\mathrm{T}}$ is in $< w_0, \ldots, w_{m-1} >$, so by the remarks above, $w_k^{\mathrm{T}} A(Ax - b) = 0$. Hence, $d_k = 0$ for all $0 \leq k \leq m-1$, showing that $Ax = b$.

The great thing about this algorithm is that the $c_{i,j}$ are zero for all $j < i-2$. To see this, note that the numerator of $c_{i,j}$ is

$$w_j^{\mathrm{T}} A^2 w_{i-1} = (Aw_j)^{\mathrm{T}} A w_{i-1} = \left(w_{j+1} + \sum_{k=0}^{j} c_{j+1,k} w_k \right)^{\mathrm{T}} A w_{i-1} = 0.$$

Thus, the Lanczos iteration simplifies to

$$w_i = A w_{i-1} - c_{i,i-1} w_{i-1} - c_{i,i-2} w_{i-2} \qquad \text{for} \qquad i \geq 2$$

where

$$c_{i,i-1} = \frac{(Aw_{i-1})^{\mathrm{T}} (Aw_{i-1})}{w_{i-1}^{\mathrm{T}} (Aw_{i-1})} \qquad \text{and} \qquad c_{i,i-2} = \frac{(Aw_{i-2})^{\mathrm{T}} (Aw_{i-1})}{w_{i-2}^{\mathrm{T}} (Aw_{i-2})}$$

For each iteration, A must be multiplied by a vector only once since the value of Aw_{i-2} is known from the previous iteration. Solving the system by standard Lanczos requires no more than n multiplies of A by a vector and $3n$ inner products. The algorithm is particularly suited for sparse matrices. If A averages d nonzero entries per column, the running time is order dn^2.

4.3 Applying Lanczos to factoring matrices

Standard Lanczos cannot be directly applied to our factoring matrices. Our matrices are neither symmetric nor positive definite. The symmetry can easily be overcome. If we call our factoring matrix B, then $A = B^{\mathrm{T}} B$ is a symmetric matrix. So we run Lanczos to solve $(B^{\mathrm{T}} B)x = B^{\mathrm{T}} b$, and a solution to this will likely be a solution to $Bx = b$ [18]. Note though that A is never computed explicitly. Instead, when we are required to multiply A by w_{i-1}, we first multiply B by w_{i-1} and then B^{T} by the

resulting product. This doubles the number of matrix-vector multiplies which is a small price to pay.

The more serious problem is the non-positive definiteness. About half of the vectors we are working with will have the property $w_j^{\mathrm{T}} A w_j = 0$ (recall we need to solve the system over the field $GF(2)$). This would cause a division by zero in computing the $c_{i,j}$. Odlyzko's solution to this is to work over the field $GF(2^r)$ where r is selected large enough so the probability of getting $c_{i,j} = 0$ is very small [18] [11]. Assuming $r \leq 32$, each field element can be stored in a computer word. Thus, the matrix-vector multiply over this field can be done in the same amount of time as doing it over $GF(2)$.

Since we are looking for null space vectors, we want to try not to get the trivial one. If we try to use Lanczos with $b = 0$, it will immediately terminate giving only $x = 0$ as the solution. A way around this is to choose a random vector y and let $b = By$. Then run Lanczos to get a solution to $Bx = b$, so that $x - y$ will be a null space vector.

An alternate solution is to use one of the block Lanczos algorithms.

In the next section we explain a simple method of distributing the computation for any iterative method that requires multiplying a matrix by a vector and then the transpose of the matrix by the result. We tried it on Montgomery's block Lanczos algorithm.

4.4 Parallel implementation of Lanczos

Most parallel computers are classified as either SIMD or MIMD. SIMD machines (single instruction multiple data) are those where the same instruction is fed to all processors at the same time. Each processor has the option of executing or not executing the instruction on its own data. MIMD machines (multiple instruction

multiple data) are those for which each processor can execute its own instructions on its own data independently of the other processors. MIMD machines give the algorithm designer a lot more freedom, but tend to be more difficult to program.

A parallel implementation of block Lanczos on a SIMD machine is described in [5]. This was on a full size MasPar, having 16,384 processors each with 64K of memory. The implementation could solve factoring matrices with up to around 2 million rows and columns in only a few days. But that was the limit. There was not enough memory to hold significantly larger matrices.

These memory limitations are the motivation for our present research. Today people are spending only a few percent (often much less) of their time on the matrix stages of factoring algorithms. Yet the problem with going on to factoring larger numbers (say, 512 bits) seems troublesome mainly because of the matrix. If we had a single, powerful computer with a very large memory all to ourselves, then a matrix of 19 million rows and columns could be solved in perhaps a couple of months. Most likely, the real time for the matrix stage would be much less than the real time for the sieving, so a couple of months would not be too long to wait for the result. However, it is unlikely that we could ever get access to a computer so big and powerful for this length of time. Certainly the memory required for this job would take up most of the memory of the computer, and therefore nobody else would be able to use it until we were finished.

So we need a new solution. Now the factoring problem is famous for using several independent workstations together on factoring one number [13]. We can ask if it is possible to do distribute the matrix problem in a similar manner. This seems harder, mainly because it requires a lot more inter-processor communication.

But in our case, the amount of *time* necessary to solve the matrix is not the most important issue. For algorithms like qs and nfs, the number of rows and columns each grow with the square root of the running times, which is super-polynomial.

While we are increasing the numbers of our computers to factor larger numbers, it is foolish to hope to solve the larger matrices on only *one* single computer. To prepare for future numbers, we need to have a way to solve the matrix without relying on the memory available of one particular computer. Thus, the first question that we should ask is, if we are willing to wait long enough, would we be able to solve the system of equations? In other words, can we gather together some collection of computers that we already have access to, and somehow make them work together on the matrix problem?

The use of multiple workstations to solve a single problem is like having a MIMD parallel computer. The computers work independently and synchronization must be enforced by the progarmmer. Software packages are available to provide communication between the machines. We used the popular package PVM, which stands for Parallel Virtual Machine.

Below we describe a parallel implementation of Montgomery's block Lanczos. We refer the reader to [17] for a complete description of the algorithm. Block Lanczos is similar to standard Lanczos, but generalizes the ideas to work with subspaces. Each subspace will have a basis of up to 32 vectors, which are combined to form a block vector. The iteration is done on these block vectors. It is not necessary to have a complete understanding of the algorithm to understand our method of parallelization. Our method also works for the standard Lanczos algorithm or any iterative method that requires multiplying the matrix by a (block) vector, and the transpose of the matrix by the result.

Our method was designed to be memory efficient. There are certainly much faster methods of parallelizing the algorithm if we are willing to duplicate large amounts of data. But again, we do not want to have to rely on each workstation involved having a large amount of memory.

We parallelize this by designating one processor as the "master" and the others as "slaves". The master will go through the block Lanczos iteration and the slaves will be responsible for the matrix-block vector multiply. The master processor may also hold part of the matrix and participate in the matrix-block vector multiply if enough memory is available. The master must have enough memory to hold the seven block vectors in the Lanczos iteration. The slaves must have enough memory to hold their portion of the matrix and only 2 block vectors.

Suppose we have P processors all equal in speed and memory. We let one be the master and the remaining $P-1$ be the slaves. If the matrix B has a total of k nonzero entries, we partition the rows of B onto the s slaves so that each slave has approximately $k/(P-1)$ nonzero entries. When the master needs the product of B^TB with block vector w_i, it will send a copy of w_i to all the slaves. The slaves will multiply their rows of B by w_i. Then each slave will multiply the transpose of their rows of B with the portion of the block vector (Bw_i) that they just computed. This will give a block vector that is a partial result. The product of $B^T \times (Bw_i)$ is obtained by summing the partial results on all slaves. This takes order $\log_2(P-1)$ communications of block vectors.

Let's take a look at the memory requirements for the RSA-130 matrix. The exact size of the matrix was 3,504,823 $\times$ 3,516,502 of density 34 (approximately 140 million nonzero entries). Since there are 4 bytes in a computer word,[2] storing the seven block vectors requires $\approx 3.5 \times 4 \times 7 = 98$ megs of memory. It takes 4 bytes to store each nonzero entry, so storing the entire matrix required $\approx 4 \times 140 = 560$ megs of memory. Thus, for a single machine to solve this system, it would need about $560 + 98 = 658$ megs of memory. This is rather large, but not impossible to find.

[2]The Cray C90 Supercomputer has 8 byte words. They took advantage of this when they originally solved the matrix, so the block vectors actually used up about 196 megs of memory.

However, it is not as difficult to find 8 workstations with 128 megabytes each. This is all that is needed for our method: the master must have 98 megs to store the 7 block vectors, and the 7 slaves each need $560/7 = 80$ megs for their portions of the matrix plus 28 megs to store the two block vectors.

Initially we tried our method on some of the matrices for the 70 and 80 digit numbers from appendix E, using four SPARC 5 workstations with communication over the Ethernet. We had the master participating in the matrix-block vector multiply, doing about the same amount of work as each of the slaves. It took almost twice as much time as it would have on a single workstation. The running time was dominated by the communication cost.

We then tried running our program on 5 processors of an IBM SP2 (MIMD) parallel computer. The SP2 processors are RS/6000's running at 66.7 MHz. They communicate through a bidirectional multistage switch of bandwidth 40 megabytes/second. We solved a 78,886$\times$81,194 factoring matrix of density ≈ 34 in about 24 minutes (real time). We also ran it on the matrix from our 116 digit number (see section 3.8). The 190,211$\times$196,701 matrix of density ≈ 40 was solved in 2 hours and 45 minutes. The communication time was about equal to the matrix-block vector multiply time.

Our method of parallelization is not the ultimate answer, but it is a starting point. It is a method that will work and is relatively easy to program. Using 8 processors on an IBM SP2, we should be possible to handle a matrix the size of RSA-130 in about one month. Adding more processors may not improve the speed because of the increased cost of communication. On the other hand, a faster switch for communication between processors may significantly increase the speed. The time required for this method make it not quite practical to handle a matrix with 19 million rows and columns.

4.5 Other ideas for parallelizing iterative methods

Our method of parallelization for block Lanczos involves constantly having to synchronize the processors after every matrix-block vector multiply. This synchronization can slow down the iteration significantly, so it would be better if we could program it so that communication happens less frequently. One way of doing this is to run the algorithm as if the word size was much larger than 32, for example 320. The number of iterations decreases by a factor of 10, but each iteration will require multiplying the matrix by the block vector which is in 10 separate chunks. So the overall speed of the Lanczos iteration is about the same (assuming the other parts of the code are still negligible, which might not be true), but synchronization happens one tenth as often.

Another idea is to try a MIMD version of the Lanczos implementation in [5].

Or, probably better than the other ideas, is to try to parallelize block Wiedemann instead. Block Wiedemann has one big advantage over block Lanczos: the fact that it doesn't require multiplication by B^{T}. This can cut down on the communication quite a bit. For block Lanczos, some processors are doing order $\theta(\log_2 P)$ communications of block vectors per iteration (P is the number of processors). But when only B is being multiplied by a block vector, each processor will get a fraction (on average, about $1/P$) of the result. These fractions must be traded among each other so all processors have the entire result. The total amount of data that needs to be communicated is $\theta(1)$ block vectors for each processor, which also can be done in $\log_2 P$ steps. Thus, with block Wiedemann, increasing the number of processors will not make as much of an impact on communication speed. For example, suppose $P = 4$ and each processor has one fourth of the matrix. Then after the local multiply, each processor will have one fourth of the result. The 4 processors must exchange their portion of the result vector with each other. This is done in two steps. In step

1, the first two processors trade their portion of the result with each other while the last two processors do the same. Now all processors have half of the result vector. In step 2, the first processor trades its half of the result with the third processor, while the second processor trades with the fourth. Now each processor has the result, and has sent and received $\frac{1}{4} + \frac{1}{2} = \frac{3}{4}$ block vectors. In general, if $P = 2^k$, then each processor sends and receives $\frac{2^k-1}{2^k}$ block vectors total. But in block Lanczos, some processors will trade k full vectors at each iteration.

Finally we remark that Odlyzko has suggested from the beginning a combination of structured Gauss and an iterative method [18]. To the author's knowledge this has not been tested yet.

Chapter 5

Conclusion

We have demonstrated that one can factor with siqs about two times faster than with mpqs alone, at least for numbers between 60 and 80 digits. Moreover, we have given data showing how we arrived at the factor of 2, so that readers can draw their own conclusion on the accuracy of our experiments.

We can think of three possible reasons why siqs was not used for the RSA-129 project. The first is that the leaders of the project were not aware that siqs is substantially faster than mpqs. We hope this thesis removes any doubt. Second, many of the machines in the RSA-129 project had very limited memory. As we remarked in section 2.1, it is not necessary to store all $2B_j a^{-1} \bmod p$ values to get a noticeable speedup over the regular mpqs. In fact, one could use a combination of siqs and mpqs for a large project like this, using mpqs only on the machines that have very small memory. Third, it was not obvious how to distribute the computation. Some very practical ideas for this were suggested in [1].

We have also suggested a method of parallelizing the block Lanczos iterative method, and tried it out. Our method was somewhat successful, but not practical for handling a matrix that might come from a nfs factorization of a 512 bit number. We should not lose hope, however, since other ideas for parallelizing iterative methods were suggested. We expect these ideas to improve the timings significantly.

Bibliography

[1] W. R. Alford, Carl Pomerance, "Implementing the self initializing quadratic sieve on a distributed network," pp. 163-174; in: A.J. van der Poorten, I. Shparlinski, and H.G. Zimmer (eds), *Number theoretic and algebraic methods in computer science*, World Scientific, 1995.

[2] D. Atkins, M. Graff, A.K. Lenstra, P.C. Leyland, "The magic words are squeamish ossifrage," Advances in Cryptology, Asiacrypt '94, Lecture Notes in Comput. Sci. 917 (1995), pp. 263-277.

[3] J. Brillhart, D.H. Lehmer, J.L. Selfridge, B. Tuckerman, 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*, Contemporary Mathematics, vol. 22, Providence: A.M.S., 1988.

[4] E.R. Canfield, P. Erdös, C. Pomerance, "On a problem of Oppenheim concerning 'factorisatio numerorum', " J. Number Theory 17 (1983), pp. 1-28.

[5] S. Contini, A.K. Lenstra, "Implementations of blocked Lanczos and Wiedemann algorithms," in preparation.

[6] D. Coppersmith, "Solving linear equations over $GF(2)$: Block Lanczos algorithm," Linear Algebra and its Applications 192 (1993), pp. 33-60.

[7] D. Coppersmith, "Solving homogeneous linear equations over $GF(2)$ via block Wiedemann algorithm," Math. Comp. 62 (1994), pp. 333-350.

[8] J. Cowie, B. Dodson, R.M. Elkenbracht, A.K. Lenstra, P.L. Montgomery, J. Zayer, "A world wide number field sieve factoring record: on to 512 bits," Advances in Cryptology, Asiacrypt '96, Lecture Notes in Comput. Sci. 1163 (1996), pp. 382-394.

[9] J.A. Davis, D.B. Holdridge, "Factorization using the quadratic sieve algorithm," Sandia Report Sand 83-1346, Sandia National Laboratories, Albuquerque, NM, 1983.

[10] B. Dodson, A.K. Lenstra, "NFS with four large primes: an explosive experiment," Advances in Cryptology, Crypto '95, Lecture Notes in Comput. Sci. 765 (1994), pp. 28-39.

[11] B.A. LaMacchia, A.M. Odlyzko, "Solving large sparse linear systems over finite fields," Advances in Cryptology, Crypto '90, Lecture Notes in Comput. Sci. 537 (1991), pp. 109-133.

[12] A.K. Lenstra, "Massively parallel computing and factoring," Proceedings Latin Amer. Symp. on Theor. Inform. '92, Lecture Notes in Comput. Sci. 583 (1992), pp. 344-355.

[13] A.K. Lenstra, M.S. Manasse, "Factoring by electronic mail," Advances in Cryptology, Eurocrypt '89, Lecture Notes in Comput. Sci. 434 (1990), pp. 355-371.

[14] A.K. Lenstra, M.S. Manasse, "Factoring with two large primes," Advances in Cryptology, Eurocrypt '90, Lecture Notes in Comput. Sci. 473 (1991), pp. 72-82.

[15] A.K. Lenstra, H.W. Lenstra, Jr. (eds), *The development of the number field sieve*, Lecture Notes in Mathematics 1554, Springer-Verlag, Berlin, 1993.

[16] J. L. Massey, "Shift-register synthesis and BCH decoding," IEEE Trans. Inform. Theory, vol. IT-15 (1969), pp. 122-127.

[17] P. L. Montgomery, "A block Lanczos algorithm for finding dependencies over $GF(2)$," Advances in cryptology, Eurocrypt '95, Lecture Notes in Comput. Sci. 921 (1995), pp. 106-120.

[18] A. M. Odlyzko, "Discrete logarithms in finite fields and their cryptographic significance," Advances in Cryptology, Eurocrypt '84, Lecture Notes in Comput. Sci. 209 (1985), pp. 224-313.

[19] C. Pomerance, "Analysis and comparison on some integer factoring algorithms," in: H.W. Lenstra , Jr., R. Tijdeman (eds), *Computational methods in number theory*, Mathematical Centre Tracts 154, 155, Mathematisch Centrum, Amsterdam, 1982, pp. 89-139.

[20] C. Pomerance, "The number field sieve," in: *Math. Comp. 1943-1993, Fifty Years of Computational Mathematics* (W. Gautschi, ed.), Proc. Symp. Appl. Math. 48, American Mathematical Society, Providence, 1994, pp. 465-480.

[21] C. Pomerance, "The quadratic sieve factoring algorithm," Advances in Cryptology, Lecture Notes in Comput. Sci. 209 (1985), pp. 169-182.

[22] C. Pomerance, J. W. Smith, "Reduction of large, sparse matrices over a finite field via created catastrophies," Experimental Math. 1 (1992), 90-94.

[23] C. Pomerance, J. W. Smith, R. Tuler, "A pipeline architecture for factoring large integers with the quadratic sieve algorithm," SIAM J. Comput. 17 (1988), pp. 387-403.

[24] R. Rivest, A. Shamir, L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," Commun. ACM 21 (1978), pp. 120-126.

[25] R. D. Silverman, "The multiple polynomial quadratic sieve," Math. Comp., 48 (1987), pp. 329-339.

[26] D. Wiedemann, "Solving sparse linear equations over finite fields," IEEE Trans. Inform. Theory 32 (1986), pp. 54-62.

Appendix A

QS analysis

Here we give a very rough heuristic proof that the quadratic sieve has running time $e^{(1+o(1))\sqrt{\log N \log\log N}}$ using the best choice of F, $e^{(\frac{1}{2}+o(1))\sqrt{\log N \log\log N}}$.

First note that to sieve m locations, the amount of time spent is $m \sum_{p \in f.b.} \frac{2}{p} \sim m \log\log F$. The residues are approximately $\sqrt{N}$. If we assume these residues are like random, the probability of finding a smooth one is $\approx u^{-u}$ where $u = \frac{\log N^{\frac{1}{2}}}{\log F}$ [4]. So the expected number of locations we need to consider to get one smooth is u^u, which takes time approximately $u^u \log\log F$. Altogether we need order $F/\log F$ smooths, so the expected running time is $T = u^u \frac{F}{\log F} \log\log F$. Note that the amount of time to do the trial division on the relations that we expect to be smooth is negligeable. We are also temporarily ignoring the time for the matrix stage of the algorithm.

We want to find the choice of F that maximizes this function, so we must compute $\frac{dT}{dF}$ and set it equal to 0.

$$\frac{dT}{dF} = u^u \frac{(-u)(1+\log u)}{F \log F} \frac{F}{\log F} \log\log F + \frac{\log F - 1}{\log^2 F}(u^u) \log\log F + \frac{1}{F \log F}(u^u) \frac{F}{\log F}$$

$$= \frac{u^u \log\log F}{\log^2 F} \left[-u(1+\log u) + (\log F - 1) + \frac{1}{\log\log F} \right].$$

For this to be zero, we must have

$$u(1+\log u) = \log F - 1 + \frac{1}{\log\log F}$$

Our claim is that $F = e^{(\frac{1}{2}+o(1))\sqrt{\log N \log\log N}}$ is the solution to this equation. To see this, note that right hand side of the equation is just $\log F = (\frac{1}{2} +$

$o(1))\sqrt{\log N \log\log N}$, since the other terms get swallowed by the $o(1)$. For the left hand side, $u = \frac{\log N^{\frac{1}{2}}}{\log F}$

$$= \frac{\frac{1}{2}\log N}{\frac{1}{2}(1+o(1))\sqrt{\log N \log\log N}}$$

$$= (1+o(1))\sqrt{\log N/\log\log N}$$

because $1/(1+o(1)) = 1+o(1)$. Also, $\log u =$

$$\log(1+o(1)) + \frac{1}{2}\log\log N - \frac{1}{2}\log\log\log N.$$

Hence, the $u(1+\log u)$ is just

$$\frac{1}{2}(1+o(1))\sqrt{\log N \log\log N},$$

again hiding the dirty stuff behind the $o(1)$. We have shown, for this choice of F, the two sides are equal. The running time is $Fe^{u\log u}\left(\frac{\log\log F}{\log F}\right)$

$$= e^{(\frac{1}{2}+o(1))\sqrt{\log N\log\log N}} e^{\frac{1}{2}(1+o(1))\sqrt{\log N\log\log N}} \left(\frac{\log\log F}{\log F}\right) = e^{(1+o(1))\sqrt{\log N\log\log N}}.$$

Of course, we also have to consider the time for the matrix stage. The matrix will have order $F/\log F$ rows and columns, but will be very sparse. Using sparse matrix methods such as [26] [7] [6] [17], this can be solved in the same time.

Let us remark that we have been a little sloppy in this derivation. The residues are actually bounded by $N^{\frac{1}{2}+\epsilon}$ where epsilon is approximately $\frac{\log T}{\log N}$. It tends to zero as N gets large. A different, perhaps more elegant, derivation of the running time is given in [20].

Appendix B

Extrapolating to 512 bits

The running times of qs and nfs are (conjecturally)

$$e^{(1+o(1))\sqrt{\log N \log\log N}}$$

and

$$e^{((64/9)^{\frac{1}{3}}+o(1))(\log N)^{\frac{1}{3}}(\log\log N)^{\frac{2}{3}}}$$

respectively. The number of rows and columns in the matrix is asymptotically the square root of the running times in both cases.

We will ignore the $o(1)$ to estimate comparisons of the matrix size and computing power requirements for factoring a 512 bit number (155 digits) as compared to 129 or 130 digit numbers.

For nfs, the amount of computing power required is

$$\approx e^{(64/9)^{\frac{1}{3}}(\log 10^{155})^{\frac{1}{3}}(\log\log 10^{155})^{\frac{2}{3}}} / e^{(64/9)^{\frac{1}{3}}(\log 10^{130})^{\frac{1}{3}}(\log\log 10^{130})^{\frac{2}{3}}} \approx 29.1$$

times the computing power for RSA-130. The matrix size would be about 5.4 times the size of the RSA-130 matrix. Multiplying this by 3.5 million gives a matrix size of nearly 19 million rows and columns.

For qs, the computing power required is

$$\approx e^{\sqrt{\log 10^{155}\log\log 10^{155}}} / e^{\sqrt{\log 10^{129}\log\log 10^{129}}} \approx 107.3$$

times the computing power for RSA-129. The matrix size would be about 10.4 times the size of the RSA-129 matrix, which would be about 5.5 million rows and columns.

Appendix C

Gray codes

We have used a Gray code to obtain all 2^{s-1} combinations of $\pm B_1 \pm \ldots \pm B_{s-1} + B_s$. The formula is:

$$\begin{aligned} b_1 &:= B_1 + \ldots + B_s \\ b_{i+1} &:= b_i + 2(-1)^{\lceil i/2^\nu \rceil} B_\nu \qquad \text{for } i = 1 \text{ to } 2^{s-1} - 1 \end{aligned} \tag{C.1}$$

where $2^\nu \parallel 2i$.

For example, suppose $s = 4$. Initially, $b_1 = +B_1 + B_2 + B_3 + B_4$. The next values of b_i are computed in the table below:

i	ν	$(-1)^{\lceil i/2^\nu \rceil}$	b_{i+1}
1	1	-1	$-B_1 + B_2 + B_3 + B_4$
2	2	-1	$-B_1 - B_2 + B_3 + B_4$
3	1	$+1$	$+B_1 - B_2 + B_3 + B_4$
4	3	-1	$+B_1 - B_2 - B_3 + B_4$
5	1	-1	$-B_1 - B_2 - B_3 + B_4$
6	2	$+1$	$-B_1 + B_2 - B_3 + B_4$
7	1	$+1$	$+B_1 + B_2 - B_3 + B_4$

Here we want to show that this actually works. In other words, for the 2^{s-1} iterations of the formula, we do get all 2^{s-1} combinations. Viewing i in binary, we will number the bits from right to left. The rightmost bit (least significant) will be referred to as the first bit, the second rightmost as the second bit, and so on. The

value of ν is then the bit number of the rightmost 1 in the binary encoding of i. The value of $(-1)^{\lceil i/2^\nu \rceil}$ depends upon whether $\lceil i/2^\nu \rceil$ is odd or even. It is even if and only if bit number $\nu + 1$ is 1. To see this, realize that dividing by 2^ν is the same as shifting the bits to the right ν places. Bit number ν (which holds the value 1) then gets shifted to the first position to the right of the binary point, and bit number $\nu + 1$ gets shifted to the rightmost bit prior to the binary point. The ceiling function rounds up, so the bit prior to the binary place gets flipped. Hence, if bit $\nu + 1$ originally was 1, then the rightmost bit of the integer $\lceil i/2^\nu \rceil$ is 0. Similarly, if bit $\nu + 1$ originally was 0, then the rightmost bit of the integer $\lceil i/2^\nu \rceil$ is 1. But the rightmost bit determines if the integer is odd or even.

We can prove that formula (C.1) goes through all combinations by an inductive argument. For $s = 2$, $b_1 = B_1 + B_2$ and $b_2 = -B_1 + B_2$. Assuming we know the formula gives us all combinations of $\pm B_1 \pm \ldots \pm B_{s-1} + B_s$ for $i = 1$ to $2^{s-1} - 1$, we want to show that it gives all combinations of $\pm B_1 \pm \ldots \pm B_s + B_{s+1}$ for $i = 1$ to $2^s - 1$. The main points to our argument are:

1. When i runs from 1 to $2^{s-1} - 1$, we get all combinations that have the last two values (B_s and B_{s+1}) positive.

2. When $i = 2^{s-1}$, B_s gets switched to negative in $b_{2^{s-1}+1}$.

3. For $i = 2^{s-1} + 1$ to $2^s - 1$, the iteration goes through the same combinations for the signs of $B_1, \ldots, B_{s-1}$ as the first $2^{s-1} - 1$ iterations, but in the reverse order. In other words, the signs of $B_1, \ldots, B_{s-1}$ in b_i will be the same as the signs of $B_1, \ldots, B_{s-1}$ in $b_{2^s - i + 1}$.

Combining these three points, $b_1, \ldots, b_{2^{s-1}}$ give the combinations $\pm B_1 \ldots \pm B_{s-1} + B_s + B_{s+1}$, and $b_{2^{s-1}+1}, \ldots, b_{2^s-1}$ give the combinations $\pm B_1 \ldots \pm B_{s-1} - B_s + B_{s+1}$, which will complete the proof.

We can assume (1) is true by the induction hypothesis. (2) follows easily by checking the computation in the iteration (C.1). To prove (3), we use another inductive argument. We know the signs of $B_1, \ldots, B_{s-1}$ for $b_{2^{s-1}}$ are the same as those for $b_{2^{s-1}+1}$ because of (2). Assume it is true for b_{i+1} and b_{2^s-i}, we want to show it also holds for b_i and b_{2^s-i+1}. We claim the change at iteration i is the same as the change at iteration $2^s - i$, but the opposite sign. In other words, if $b_{i+1} = b_i + 2B_\nu$ then $b_{2^s-i+1} = b_{2^s-i} - 2B_\nu$, and if $b_{i+1} = b_i - 2B_\nu$, then $b_{2^s-i+1} = b_{2^s-i} + 2B_\nu$ $(1 \leq i \leq 2^{s-1} - 1)$. It is not difficult to see that $2^\nu \parallel 2i$ if and only if $2^\nu \parallel 2(2^s - i)$. To show the signs are opposite for B_ν, we show bit number $\nu + 1$ of i is different than bit number $\nu + 1$ of $2^s - i$. We know bit number ν is 1 in both cases, and we know the sum of i and $2^s - i$ has all 0 bits except bit number $s + 1$. Since i has at most $s - 1$ bits, bit $\nu + 1$ in the sum must be a zero, and there is a carry of 1 from the sum at bit ν, which means the sum in the $\nu + 1$ position must be a 1. Hence one of the bits is 1 and the other is 0.

Completing the proof, if $b_{i+1} = b_i + 2B_\nu$ then $b_i = b_{i+1} - 2B_\nu$. The signs of $B_1, \ldots, B_{s-1}$ for b_{i+1} are the same as the signs for b_{2^s-i}, by induction. So the signs for b_i are going to be the same as for b_{2^s-i}, except B_ν becomes negative. But the same is true for b_{2^s-i+1} since $b_{2^s-i+1} = b_{2^s-i} - 2B_\nu$. The case for $b_{i+1} = b_i - 2B_\nu$ and $b_{2^s-i+1} = b_{2^s-i} + 2B_\nu$ is proved similarly.

Appendix D

Timings

All times listed are of the form minutes:seconds.

M = 600,000 F = 60,000			
BLOCKSIZE	*number smooths*	*siqs time*	*mpqs time*
600,000	100	n.a.	2:37
300,000	100	n.a.	2:34
200,000	100	n.a.	2:26
100,000	100	n.a.	2:26
50,000	100	n.a.	2:28

Table D.1: **cache experiment for first 60 digit number**

BLOCKSIZE = 100,000 F = 60,000 3,101 primes in f.b.				
M	*siqs time*	*(smooths)*	*mpqs time*	*(smooths)*
1,500,000	n.a.		2:38	(100)
1,000,000	n.a.		2:40	(101)
700,000	n.a.		2:53	(100)
600,000	1:52	(110)	2:26	(100)
500,000	n.a.		2:37	(100)
400,000	1:35	(116)	3:04	(100)
300,000	1:43	(106)	2:54	(100)
200,000	1:11	(102)	3:15	(100)
100,000	1:13	(110)	4:59	(100)

Table D.2: **siqs vs mpqs for first 60 digit number**

BLOCKSIZE = 100,000 F = 60,000		
program	M	*time to get* $\geq$ *1,700 smooths*
mpqs	600,000	40:33
mpqs	500,000	40:35
siqs	200,000	19:20
siqs	100,000	20:45

Table D.3: **total time to complete factorization, first 60 digit number**

BLOCKSIZE =150,000 F = 60,000 2,964 primes in f.b.				
M	*siqs time*	*(smooths)*	*mpqs time*	*(smooths)*
1,500,000	n.a.		4:07	(100)
1,350,000	n.a.		3:07	(100)
1,200,000	n.a.		2:46	(100)
1,050,000	n.a.		3:08	(100)
900,000	n.a.		2:47	(100)
750,000	n.a.		2:56	(101)
600,000	2:10	(115)	3:07	(101)
300,000	1:47	(101)	4:23	(100)
150,000	1:34	(102)	n.a.	

Table D.4: **siqs vs mpqs for second 60 digit number**

F = 60,000 3,045 primes in f.b.					
M	*BLOCKSIZE*	*siqs time*	*(smooths)*	*mpqs time*	*(smooths)*
6,000,000	150,000	n.a.		15:15	(300)
4,500,000	150,000	n.a.		14:24	(300)
3,000,000	150,000	n.a.		12:02	(300)
2,100,000	150,000	n.a.		11:50	(300)
1,500,000	150,000	n.a.		12:57	(300)
1,200,000	150,000	n.a.		10:30	(300)
900,000	150,000	n.a.		11:18	(302)
600,000	150,000	8:22	(317)	11:50	(300)
300,000	150,000	6:08	(303)	12:01	(300)
200,000	100,000	6:15	(304)	n.a.	
150,000	150,000	5:30	(303)	17:05	(300)
100,000	100,000	5:01	(300)	n.a.	

Table D.5: **siqs vs mpqs for third 60 digit number**

F	primes in f.b.	best M	mpqs expected time without large primes
150,000	6,814	1,500,000	1840.5 min
200,000	8,870	3,300,000	1438.8 min
300,000	12,876	4,500,000	1111.2 min
350,000	14,796	6,000,000	1030.5 min
500,000	20,512	5,400,000	975.2 min
600,000	24,383	4,500,000	1035.3 min
800,000	31,845	6,600,000	1060.2 min

Table D.6: **factor base experiment for first 70 digit**

F = 350,000 14,796 primes in f.b.					
M	*BLOCKSIZE*	*siqs time*	*(smooths)*	*mpqs time*	*(smooths)*
9,000,000	200,000	n.a.		91:41	(1201)
8,000,000	200,000	n.a.		88:45	(1201)
6,600,000	200,000	n.a.		91:27	(1201)
6,000,000	300,000	n.a.		85:40	(1201)
6,000,000	200,000	n.a.		83:50	(1201)
6,000,000	150,000	n.a.		84:59	(1201)
5,400,000	200,000	n.a.		88:57	(1200)
4,200,000	200,000	n.a.		87:52	(1201)
3,000,000	200,000	n.a.		89:16	(1200)
2,400,000	200,000	n.a.		86:39	(1201)
1,200,000	200,000	n.a.		101:24	(1200)
900,000	150,000	n.a.		106:02	(1200)
200,000	200,000	39:06	(1206)	n.a.	
150,000	150,000	42:44	(1206)	n.a.	

Table D.7: **siqs vs mpqs for first 70 digit number**

F = 350,000 15,007 primes in f.b.					
M	*BLOCKSIZE*	*siqs time*	*(smooths)*	*mpqs time*	*(smooths)*
9,200,000	200,000	n.a.		152:28	(1200)
8,000,000	200,000	n.a.		153:44	(1200)
7,200,000	200,000	n.a.		144:34	(1200)
6,400,000	200,000	n.a.		144:15	(1200)
5,600,000	200,000	n.a.		148:11	(1201)
4,800,000	200,000	n.a.		148:57	(1201)
4,400,000	200,000	n.a.		143:44	(1200)
4,000,000	200,000	n.a.		134:18	(1200)
3,600,000	300,000	n.a.		143:35	(1200)
3,600,000	200,000	n.a.		138:46	(1200)
3,600,000	150,000	n.a.		143:22	(1200)
3,200,000	200,000	n.a.		139:55	(1200)
2,800,000	200,000	n.a.		148:00	(1200)
2,400,000	200,000	n.a.		159:54	(1200)
2,000,000	200,000	n.a.		151:53	(1201)
1,600,000	200,000	n.a.		147:09	(1200)
1,200,000	200,000	n.a.		160:49	(1200)
400,000	200,000	76:07	(1205)	n.a.	
200,000	200,000	64:43	(1207)	n.a	

Table D.8: **siqs vs mpqs for second 70 digit number**

F = 350,000 15,088 primes in f.b.					
M	*BLOCKSIZE*	*siqs time*	*(smooths)*	*mpqs time*	*(smooths)*
15,000,000	200,000	n.a.		50:02	(1500)
9,600,000	200,000	n.a.		46:31	(1502)
8,000,000	200,000	n.a.		47:53	(1502)
7,200,000	200,000	n.a.		45:36	(1504)
6,400,000	200,000	n.a.		45:35	(1502)
5,600,000	200,000	n.a.		45:42	(1500)
4,800,000	200,000	n.a.		44:23	(1501)
4,000,000	200,000	n.a.		46:50	(1501)
3,600,000	200,000	n.a.		46:17	(1501)
3,200,000	200,000	n.a.		43:25	(1500)
2,800,000	200,000	n.a.		43:58	(1501)
2,400,000	200,000	n.a.		45:13	(1501)
2,000,000	200,000	n.a.		46:17	(1500)
1,000,000	200,000	n.a.		51:09	(1500)
200,000	200,000	22:18	(1516)	n.a.	
150,000	150,000	23:01	(1516)	n.a.	

Table D.9: **siqs vs mpqs for third 70 digit number**

F	primes in f.b.	best M	mpqs expected time, no large primes
750,000	30,112	6,000,000	3428.8 min
900,000	35,487	6,000,000	3252.1 min
1,000,000	39,059	6,000,000	3184.7 min
1,200,000	46,228	6,000,000	3300.1 min
1,500,000	56,887	6,000,000	3404.9 min

Table D.10: **factor base experiment for first 80 digit number**

F = 900,000 35,487 primes in f.b.					
M	*BLOCKSIZE*	*siqs time*	*(smooths)*	*mpqs time*	*(smooths)*
15,000,000	300,000	n.a.		281:13	(2803)
12,000,000	300,000	n.a.		268:02	(2801)
9,000,000	300,000	n.a.		272:18	(2804)
7,800,000	300,000	n.a.		259:37	(2802)
7,200,000	300,000	n.a.		264:42	(2800)
6,600,000	300,000	n.a.		259:04	(2800)
6,000,000	400,000	n.a.		268:27	(2800)
6,000,000	300,000	n.a.		256:36	(2800)
6,000,000	200,000	n.a.		259:54	(2800)
5,400,000	300,000	n.a.		273:25	(2800)
4,200,000	300,000	n.a.		270:58	(2800)
3,600,000	300,000	n.a.		280:01	(2800)
3,000,000	300,000	n.a.		271:16	(2800)
300,000	300,000	125:17	(2861)	n.a.	
300,000	150,000	129:38	(2861)	n.a.	
200,000	200,000	124:25	(2858)	n.a.	

Table D.11: **siqs vs mpqs for first 80 digit number**

F = 900,000 35,559 primes in f.b.					
M	*BLOCKSIZE*	*siqs time*	*(smooths)*	*mpqs time*	*(smooths)*
15,000,000	300,000	n.a.		401:22	(2800)
12,000,000	400,000	n.a.		394:56	(2801)
12,000,000	300,000	n.a.		387:40	(2801)
12,000,000	200,000	n.a.		388:23	(2801)
9,000,000	300,000	n.a.		395:21	(2801)
7,800,000	300,000	n.a.		389:32	(2800)
7,200,000	300,000	n.a.		375:58	(2800)
6,600,000	300,000	n.a.		385:03	(2800)
6,000,000	400,000	n.a.		397:59	(2800)
6,000,000	300,000	n.a.		390:15	(2800)
6,000,000	200,000	n.a.		393:06	(2800)
5,400,000	300,000	n.a.		383:03	(2800)
4,200,000	300,000	n.a.		403:07	(2800)
3,000,000	300,000	n.a.		415:30	(2800)
300,000	300,000	175:59	(2801)	n.a.	
200,000	200,000	183:44	(2838)	n.a.	

Table D.12: **siqs vs mpqs for second 80 digit number**

F = 900,000 35,650 primes in f.b.					
M	*BLOCKSIZE*	*siqs time*	*(smooths)*	*mpqs time*	*(smooths)*
15,000,000	300,000	n.a.		536:08	(2802)
12,000,000	300,000	n.a.		523:58	(2800)
9,000,000	600,000	n.a.		512:36	(2801)
9,000,000	300,000	n.a.		498:03	(2801)
9,000,000	200,000	n.a.		511:48	(2801)
8,400,000	300,000	n.a.		513:45	(2800)
7,800,000	300,000	n.a.		498:42	(2800)
7,200,000	300,000	n.a.		502:25	(2800)
6,000,000	300,000	n.a.		504:59	(2801)
5,400,000	300,000	n.a.		513:46	(2801)
2,400,000	300,000	n.a.		535:51	(2800)
1,200,000	300,000	n.a.		679:42	(2800)
600,000	300,000	248:39	(2807)	n.a.	
400,000	400,000	236:53	(2818)	n.a.	
400,000	200,000	234:54	(2818)	n.a.	
300,000	300,000	226:26	(2816)	n.a.	
300,000	150,000	233:04	(2816)	n.a.	
200,000	200,000	229:41	(2804)	n.a.	

Table D.13: **siqs vs mpqs for third 80 digit number**

Appendix E

Full factorization tables

$F = 350{,}000$ 14,796 primes in f.b.						
	M	*BLOCKSIZE*	*smooths*	*partials*	*matches*	*time*
mpqs	6,000,000	200,000	7,501	109,386	10,669	546:15
siqs	200,000	200,000	7,511	98,730	9,175	246:43

Table E.1: **full factorization data, first 70 digit number**

$F = 350{,}000$ 15,007 primes in f.b.						
	M	*BLOCKSIZE*	*smooths*	*partials*	*matches*	*time*
mpqs	4,000,000	200,000	7,500	117,317	11,953	917:28
siqs	200,000	200,000	7,508	110,113	10,882	411:58

Table E.2: **full factorization data, second 70 digit number**

$F = 350{,}000$ 15,088 primes in f.b.						
	M	*BLOCKSIZE*	*smooths*	*partials*	*matches*	*time*
mpqs	3,200,000	200,000	7,504	83,260	7,499	211:53
siqs	200,000	200,000	7,707	80,969	7,036	112:06

Table E.3: **full factorization data, third 70 digit number**

F = 900,000 35,487 primes in f.b.						
	M	$BLOCKSIZE$	*smooths*	*partials*	*matches*	*time*
mpqs	6,000,000	300,000	18,102	189,800	16,914	1719:47
siqs	200,000	200,000	18,231	190,737	16,453	801:25

Table E.4: **full factorization data, first 80 digit number**

F = 900,000 35,559 primes in f.b.						
	M	$BLOCKSIZE$	*smooths*	*partials*	*matches*	*time*
mpqs	7,200,000	300,000	18,002	224,497	20,655	2502:37
siqs	300,000	300,000	18,018	211,070	18,694	1188:25

Table E.5: **full factorization data, second 80 digit number**

F = 900,000 35,650 primes in f.b.						
	M	$BLOCKSIZE$	*smooths*	*partials*	*matches*	*time*
mpqs	9,000,000	300,000	18,000	241,410	22,679	3232:24
siqs	300,000	300,000	18,041	221,369	19,553	1446:01

Table E.6: **full factorization data, third 80 digit number**