# BOUND OPTIMIZATION FOR PARALLEL QUADRATIC SIEVING USING LARGE PRIME VARIATIONS

---

An Honors Thesis

Presented to

The Faculty of the Department of Computer Science

Washington and Lee University

In Partial Fulfillment Of the Requirements for

Honors in Computer Science

---

by

Andrew Granville West

May 2007

*The problem of distinguishing prime numbers from composite numbers and of resolving the latter into their prime factors is known to be of the most important and useful in arithmetic. It has engaged the industry and wisdom of ancient and modern geometers to such an extent it would be superflous to discuss the problem at length. Nevertheless... the dignity of the science itself seems to require that every possible means be explored for the solution of a problem so elegant and so celebrated.*

– *Karl Gauss*

# Contents

# ACKNOWLEDGMENTS

This work would not have been possible without the contributions of several individuals who I would like to recognize here. First and foremost, my advisor Rance Necaise, who supported my interest in a topic probably outside of his own research areas. The remainder of the Computer Science department; Ken Lambert, Simon Levy, and Thomas Whaley were instrumental because of the education they endowed both in and out of the classroom.

Similar in function was the Mathematics department, to which I must apologize for my inability to complete their major. I would like to give special mention to Wayne Dymacek and Jacob Siehler, whose passion for mathematics is nothing less than inspiring.

The team of Olof Åsbrink and Joel Brynielsson from the Royal Institute of Technology in Sweden graciously sent their Quadratic Sieve source code, which was critical in troubleshooting my own efforts.

Finally, at a personal level, I must thank my parents, biological brothers, and brothers of Sigma Nu fraternity for all the support (or distraction, in the latter case) that they provided.

# ABSTRACT

The Quadratic Sieve (QS) factorization algorithm is a powerful means to perform prime decompositions that combines number theory, linear algebra, and brute processing power. Created by Carl Pomerance in 1985, it is the second fastest general purpose factorization method as of this writing, behind only the Number Field Sieve.

We describe an efficient QS implementation which is accessible to an undergraduate audience. The majority of papers on this topic rely on complex mathematical notation as their primary means of explanation. Instead, we attempt to combine math, discussion, and examples to promote understanding. Additionally, few authors ever present implementation level detail. With the significant time and memory requirements of the extremely iterative algorithm, even minor optimizations can result in large runtime improvements and these are described. Discussion covers both sequential and parallel implementations, as the latter is required to factor numbers of a signficant magnitude.

We use our implementation to optimize the selection of prime bounds which are critical to QS runtime. Analysis begins with basic QS, a simplified version of the more powerful PQS (QS w/Large Prime Variations), which is later given attention. Both theory and concrete statistics guide us in determining how variable selection can be used to minimize runtimes. Given that data, we attempt to extrapolate bound selection for larger integers, whose size makes them inappropriate for casual experimentation. We find that predicting ideal bounds for basic QS can be done with some degree of accuracy, however, PQS behaves more erratically making extrapolation difficult.

# BOUND OPTIMIZATION FOR PARALLEL QUADRATIC SIEVING USING LARGE PRIME VARIATIONS

# Chapter 1

# Introduction

The problem of breaking down a composite number into its unique prime factors is one that has puzzled mathematicians for over two millenia. Indeed, it was around 300 B.C. that Euclid laid the basis for the art in his famous text, *Elements*. In the book, Euclid states a lemma that Carl Gauss would formalize and prove in 1801, the Fundamental Theorem of Arithmetic (FTOA).

The FTOA states that every natural number greater than 1 has a unique representation as a product of primes [10]. The purpose of prime factorization (alternatively, prime decomposition) is to determine this representation. Integer factorization is an important practice because its presumed difficulty forms the basis for several important cryptography systems, including one securing the majority of e-commerce transactions. The algorithm discussed herein is one of the most efficient means known to perform prime factorizations.

## 1.1 The Quadratic Sieve

The Quadratic Sieve (QS) is an algorithm developed by number theorist Carl Pomerance in 1981 [8]. At the time of its invention it was the fastest known general purpose decomposition method. QS combines number theory, linear algebra, and brute force processing power to produce factorizations. Runtime of the method is primarily dependent on the size of the

input to be factored, not on any special mathematical characteristics it may have.

Generally speaking, QS is a means by which to factor large composite integer $N$. QS is most effective when $N$ is a semiprime, the product of just two primes. It is important that these two factors be approximately the same size. If $N$ is small in size or contains small factors more trivial decomposition methods become appropriate.

The largest factorization ever completed by the QS method was of RSA-129, a 129 decimal digit semiprime. Its decomposition in 1994 took eight months across 600 computers in a distributed computing fashion similar to that of the SETI@Home project. When the challenge number was first published in 1977, security experts estimated it would take 40 quadrillion years to factor using the technology of the time [6].

## 1.2 Modern Factorization Methods

In the current state of the art only three factorization methods necessitate discussion. The Quadratic Sieve (QS), Elliptic Curve Method (ECM), and General Number Field Sieve (GNFS) are the only known algorithms with sub-expotential time complexity [3].

Lenstra's ECM is a method which has the same asymptotic time complexity bound as QS, but it serves a specialized purpose. While ECM does operate on large $N$, its running time is dependent not on the size of $N$, but the size of the factors of $N$. It is most appropriate for finding moderately sized divisors (less than 25 digits) and therefore is ineffective for large semiprimes. Furthermore, dependency on floating point arithmetic slows the algorithm in practice.

QS is the focus of our discussion and will be covered in great detail in later chapters. However, it should be mentioned that a number of improvements have been made to the original algorithm. Common variations include multiple polynomials (MPQS), self-initialization (SIQS), large prime variations (PQS), and double large prime variations (PPQS). At the highest level, these are often combined to create the most efficient implementation possible (the PPSIMPQS: *double large prime variation self initializing multiple polynomial*

*Quadratic Sieve* is quite the factoring mechanism!). We will concern ourselves only with the basic (single polynomial) QS model with single large prime variations. Though not the fastest method, this selection permits appropriate focus on the research question at hand.

QS serves as the predecessor to GNFS. GNFS takes the basic idea of QS and improves efficiency by using higher-order polynomials and algebraic rings. GNFS was also developed by Carl Pomerance and is the fastest known factorization method as of this writing. Even using supercomputers and across the largest parallel environments, the largest general-purpose factorization ever was in 2005 of RSA-200, a 200 deciminal digit semiprime that took 18 months to factor on a dedicated 80 node cluster [7]. Because of its high overhead, GNFS is slower than QS for input less than approximately 100 decimal digits.

## 1.3 Motivation

The motivation behind our work is to describe an optimal means by which two independent variables (the smoothness-bound and large-prime-bound) critical to QS runtime can be chosen. Since runtime data will be central to this analysis it is important that an efficient implementation of QS be used. Futhermore, creating a standard/characteristic QS will allow application of our findings to those implementations already in existence. A parallel version is necessary to perform a sufficient number of signficantly sized test cases in a reasonable time frame.

Therefore, another objective is creating and describing such an implementation. QS has been the subject of an abundance of prior research. However, discussions of implementation strategy are weak at best. Authors who do present QS in some low-level detail, such as Åsbrink and Brynielsson [1], make no claims concerning the efficiency of their work. The majority of other authors rely only on a mathemetically obfuscated discussion to present QS. Often, these presentations are so complex and abstract that they are inaccessible to the average undergraduate student.

Coding a sub-optimal but functioning QS is a straightforward process, but QS is an

extremely iterative method so even the smallest optimizations can produce dramatic reductions in runtime. Similarly, optimal independent variable selection is critical when dealing with an algorithm of near expotential complexity. Though the majority of current research has moved to more advanced methods such as MPQS and GNFS, the similarity among methods allows this work to remain viable.

## 1.4 Thesis Layout

To provide the basis of our work, the mathematical foundations of QS must be established with discussion concerning trial division, Fermat, and congruence of squares. This foundation permits an explanation of Dixon's algorithm, the predecessor to QS. An explanation of quadratic residues and their application to Dixon's method will yield an understanding of QS. Finally, introducing large prime variants will conclude the abstract mathematical discussion in Chapter 2.

Next, sequential implementation details will be addressed in Chapter 3. Well established speedup strategies along with more subtle coding decisions will be covered. Type choices, allocations, and specific methods will be discussed when non-trivial. The same discussion will then take place with regard to a parallel environment in Chapter 4.

Usage of this mathematical theory and implementation detail will permit prudent analysis of prime bound selection. A large number of test runs will be summarized, and the regression of these data sets will allow extrapolation about larger factorizations. Several such tests will be conducted to access the accuracy of our findings. Timings and associated graphs which support our claims are introduced in Chapter 5.

The paper will then conclude in Chapter 6 by detailing the significance of integer factorization, especially with regard to the RSA public-key encryption system. Finally, mention will be made of the possible shortcomings of this work and the hardware on which it was performed.

# Chapter 2

# Mathematical Background

The Fundamental Theorem of Arithmetic was introduced in the previous chapter as the basis for prime factorizations. Expanding on this, attention will be given to the mathemetical preliminaries of the Quadratic Sieve (QS) and eventually QS itself.

## 2.1 Trial Division

The most naive and simplistic method by which prime factors can be determined is known as trial division. Given positive integer $N$, trial division attempts brute-force division for every prime on the interval $2 \ldots \sqrt{N}$. The algorithm terminates once an even divisor is found or the upper bound is reached (and thus $N$ is prime).

If $N$ contains trivially small factors, the method is very quick[1]. As $N$ and its factors grow larger, the inefficiency of division at the machine level and the large overhead of finding primes makes trial division inefficient.

[1]We should note that such $N$ are relatively common but are of no interest to the factoring community. For example, 50% of integers have 2 as a factor, 75% have either 2 or 3, and 88% have a factor under 100.

## 2.2 Fermat and Congruence of Squares

By the 17th century, Fermat had discovered that factorizations could be represented as a difference of squares. Given input $N$, Fermat's factorization method attempts to find $a$ and $b$ such that:

$$N = a^2 - b^2 = (a + b)(a - b). \tag{2.1}$$

It can be extremely difficult to find such $a$ and $b$, in some cases, more steps are required than with trial divison. In practice, the equality operator of (2.1) is much too constricting. Thus, the the weaker congruence of squares is introduced:

$$a^2 \equiv b^2 \mod N \text{ , where } a \not\equiv \pm b. \tag{2.2}$$

With some simple modular algebra, one can modify this to observe:

$$a^2 - b^2 \equiv (a + b)(a - b) \equiv 0 \mod N, \tag{2.3}$$

which implies there is a good probabability that

$$\gcd(a \pm b, N) \tag{2.4}$$

are the factors of $N$ [10]. The efficient finding of such $a$ and $b$ is the basis for many modern factorization algorithms, including Dixon's and QS.

It is important to note that because the congruence of squares is a far weaker constraint than Fermat's equality, there is some probability that the greatest common demoninator (GCD) will produce trivial results (1 or $N$ as opposed to significant factors). The probability of two integers being co-prime (and thus producing trivial factors) is a complex mathematical construct. However, mathematicians have shown that for integers $c$, $d$:

$$Prob(\gcd(c, d) = 1) = \frac{1}{\zeta(2)} = \frac{6}{\pi^2} = 0.607927102 \text{ as } c, d \to \infty \tag{2.5}$$

where $\zeta$ is the Riemann zeta function [10]. Thus, in the *worst* case one has about a 40% probability that the GCD will return non-trivial factors. Generally, this percentage is

slightly higher in practice and the majority of sources cite it to be around 50%. Due to this high failure rate, advanced methods focus on finding multiple $a$, $b$ pairs so success is probablistically ensured.

## 2.3 Dixon's Method

Dixon's factorization method is one means to find $a$, $b$ which satisfy a congruence of squares modulo $N$. Dixon's method will be presented in a mathematically abstract manner, followed by a concrete example to solidfy understanding.

### 2.3.1 Mathematical Explanation

To begin, one selects a variable, $B$, known as the smoothness bound. From this a set of all primes less or equal to $B$ is created, known as the smooth or factor base ($\{FB\}$). Using the polynomial

$$f(x) = x^2 \mod N \text{ , where } x > \sqrt{N} \text{ and } x \in \mathbb{N} \tag{2.6}$$

many $x_i$ are tested for the condition that $f(x_i)$ factors completely over the factor base. Or mathematically, one looks for $f(x_i)$ that are $\{FB\}$-smooth. Pairs ($x_i'$, $f'(x_i)$) which fulfill this criteria are known as relations and stored until *sufficient* relations are found.

One proceeds by finding a subset of $f'(x_i)$ whose product is a perfect square, $P_1$. The corresponding subset of $x_i'$ can then be multiplied to form a second product, $P_2$. Because of the nature of generating function (2.6) it can be shown that $P_1 \equiv P_2^2 \mod N$, and thus a congruence of squares.

Linear algebra is used to produce the perfect square subsets of $f'(x_i)$. For each relation, one constructs a vector of length $|\{FB\}|$ containing the exponent of each {FB} element in the factorization of $f'(x_i)$. Once mod 2 is performed on the vector, it is known as the *modulo 2 exponent vector*. For example, if $\{FB\} = \{p_1, p_2, p_3 \dots p_n\}$ and $f'(x_i) = p_1^0 + p_2^1 + p_3^3 + \dots p_n^2$ then exponent vector $V = [\ v_1\ v_2\ v_3\ \dots\ v_n\ ] = [\ 0\ 1\ 3\ \dots\ 2\ ]$ which modulo 2 is $= [\ 0\ 1\ 1\ \dots\ 0\ ]$.

We want to find a combination of exponent vectors which will produce a zero vector. A zero vector corresponds to a perfect square because even powers (0 modulo 2) of any integer (or product of integers) are trivially just squares of themselves. Each unique zero vector composition will generate a different congruence of squares. By the laws of linear algebra, one must find $|\{FB\}| + 1$ relations to ensure such a dependency.

To solve for zero vectors, insert all $f'(x_i)$ exponent vectors as the rows into a matrix and augment the identity. Consider the matrix produced when $m$ relations are found for a factor-base containing $n$ elements:

$$[\ V \quad | \quad I\ ] = \begin{bmatrix} v_{11} & v_{12} & v_{13} & \dots & v_{1n} \\ v_{21} & v_{22} & v_{23} & \dots & v_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v_{m1} & v_{m2} & v_{m3} & \dots & v_{mn} \end{bmatrix} | \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}.$$

Perform Gaussian elimination[2] in $\mathbb{F}_2$ to solve the linear system. Because Gauss reduces matrices to reduced row echelon form, all zero rows will reside at the bottom of the matrix. Post-elimination, the matrix now has a significantly different representation:

$$[\ V' \quad | \quad I'\ ] = \begin{bmatrix} v'_{11} & v'_{12} & v'_{13} & \dots & v'_{1n} \\ v'_{21} & v'_{22} & v'_{23} & \dots & v'_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v'_{m1} & v'_{m2} & v'_{m3} & \dots & v'_{mn} \end{bmatrix} | \begin{bmatrix} i_{11} & i_{12} & \dots & i_{1m} \\ i_{21} & i_{22} & \dots & i_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ i_{m1} & i_{m2} & \dots & i_{mm} \end{bmatrix}.$$

Formally, a zero row is a row residing in $V'$ containing only 0s. The adjacent $I'$ rowspace need not contain all 0's; it is this rowspace which is the key to unlocking the congruence(s). It tells the combination of exponent rows necessary to form a zero row. For each zero row, $a$, $b$ are built such that:

$$\begin{gathered} a = \prod x_k \text{ and } b^2 = \prod f(x_k) \text{ for all } i_{*k} = 1 \text{ where } * = \text{ zero row, and} \\ (x_k, f(x_k)) \text{ is relation corresponding to k-th row of the original matrix (V).} \end{gathered} \tag{2.7}$$

For each zero row one will have unique $a$ and $b$ values that fulfill the congruence of squares modulo $N$. Just as before, $\gcd(a \pm b, N)$ have a decent probability of being non-trivial factors of $N$, and the algorithm is complete.

[2] In fact, any matrix solver would work fine. Gaussian elimination is just the most familiar of such methods. As we will see, Gauss has a very efficient implementation in $\mathbb{F}_2$ that makes it appropriate for surprisingly large matrices.

### 2.3.2 Concrete Example Using Dixon

Because notation and formal mathematical definitions do not necessarily result in clear understanding, a concise example will be completed:

Factoring N = 1633 using smoothness bound B = 10
produces FB = {2, 3, 5, 7}, and the relation-search begins:

$$\begin{array}{rcrlc}
f(41) & = & 48 & = 2^4 * 3^1 * 5^0 * 7^0 & = [\ 0\ \ 1\ \ 0\ \ 0\ ] \\
f(43) & = & 216 & = 2^3 * 3^3 * 5^0 * 7^0 & = [\ 1\ \ 1\ \ 0\ \ 0\ ] \\
f(45) & = & 392 & = 2^3 * 3^0 * 5^0 * 7^2 & = [\ 1\ \ 0\ \ 0\ \ 0\ ] \\
f(47) & = & 576 & = 2^6 * 3^2 * 5^0 * 7^0 & = [\ 0\ \ 0\ \ 0\ \ 0\ ] \\
f(49) & = & 768 & = 2^8 * 3^1 * 5^0 * 7^0 & = [\ 0\ \ 1\ \ 0\ \ 0\ ]
\end{array}$$

Combine these relations into a single matrix and augment the identity:

$$\left[\begin{array}{cccc}
0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{array}\right] \Bigg| \left[\begin{array}{ccccc}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{array}\right]$$

Now perform Gaussian Elimination on the matrix:

$$\left[\begin{array}{cccc}
1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{array}\right] \Bigg| \left[\begin{array}{ccccc}
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1
\end{array}\right]$$

This yields three zero rows, which we will label $R_1 - R_3$ from the top down, respectfully. Processing each row using equation 2.7:

$$\begin{array}{lcccccccc}
R_1 : a = & 41 * 43 * 45 & = & 79335 & \text{and} & b = & \sqrt{48 * 216 * 392} & = & 2016 \\
R_2 : a = & 47 & = & 47 & \text{and} & b = & \sqrt{576} & = & 24 \\
R_3 : a = & 41 * 49 & = & 2009 & \text{and} & b = & \sqrt{48 * 768} & = & 192
\end{array}$$

Finally, one can use GCD to find the factors, trivial or otherwise:

$$\begin{array}{l}
R_1 : \gcd(a + b, N) = 23 \quad \text{and } \gcd(a - b, N) = 71 \\
R_2 : \gcd(a + b, N) = 71 \quad \text{and } \gcd(a - b, N) = 23 \\
R_3 : \gcd(a + b, N) = 71 \quad \text{and } \gcd(a - b, N) = 23
\end{array}$$

Luckily, for this particular number all three zero rows happen to produce non-trivial solutions. With multiplication one can verify that indeed $23 * 71 = 1633$, confirming the factors have been successfully found.

## 2.4 The Quadratic Sieve

QS makes several minor changes to Dixon's algorithm which dramatically increase the efficiency. Wheras Dixon uses brute force to try to factor each $f(x_i)$ over prime-base elements, QS uses a more elegant technique known as *sieving*. At the heart of this technique is a construct of some mathematical complexity: the quadratic residue.

### 2.4.1 Quadratic Residues

By definition, a number $N$ is called a quadratic residue modulo $p$ if

$$\exists r \text{ such that } N \equiv r^2 \mod p. \tag{2.8}$$

Because of the circularity of modular arithmetic, if such $r$ does exist, it will occur on the interval

$$0 < r < \frac{p}{2} \text{ for r } \in \mathbb{N}. \tag{2.9}$$

In addition, if an $r$ can be found to meet this criteria, a second solution, the trivial $r' = p - r$ will also fulfill the requirements. In factoring applications, the solution values $r$ and $r'$ are as important as the existence of a quadratic residue itself.

An efficient method for the calculation of quadratic residues remains a major unsolved problem in mathematics. If a quick deterministic algorithm existed it would allow for an integer factorization algorithm of similar complexity and vice versa. Thankfully, QS only utilizes quadratic residue solves for numbers much smaller than the $N$ being factored.

A construct called the Legendre symbol, a special case of Jacobi symbol, is often used in discussions of quadratic residues. Given odd prime $p$ and integer $N$ the Legendre symbol is defined to be:

$$\left(\frac{N}{p}\right) = \begin{cases} 0, & \text{if } p \text{ divides } N \text{ , otherwise:} \\ 1, & \text{if } N \text{ is a quadratic residue modulo } p \\ -1, & \text{if } N \text{ is a quadratic non-residue modulo } p \end{cases} \tag{2.10}$$

It should be noted that much like the complexity relationship between primality testing and factorization, determining if a residue exists is far more efficient than actually computing the residue solve value.

### 2.4.2 QS and Quadratic Residues

The application of quadratic residues to Dixon's method effectively gives one a QS. As before, one selects a smoothness bound $B$, however, the factor base definition is now stricter:

$$\{FB\} = \{p \mid p \text{ is prime} \leq B \text{ and } \left(\frac{N}{p}\right) = 1\} \tag{2.11}$$

Recall function $f(x)$ as it was defined in equation (2.6) for Dixon's method. The function used by QS focuses on finding a simpler subset of these values:

$$f(x) = x^2 - N \text{ where } \sqrt{N} < x < \sqrt{2N} \text{ and } x \in \mathbb{N}. \tag{2.12}$$

By better defining the interval on $x$, the modulus operator can be done away with altogether. This is beneficial because modulus predicates are inherently costly since they involve division. However, because equality is a subset of congruence and $x_i^2 - N \equiv x_i^2 \mod N$, the fundamental congruences that Dixon demonstrated still hold.

Most importantly, the quadratic residue solution values $r$ and $r'$ for every $p \in \{FB\}$ tell for which $f(x_i)$ that $p$ is a factor:

$$\begin{array}{c}\text{If } p_n \in \{FB\} \text{ and } r_n, r'_n \text{ are residue sols. then } p_n \text{ divides } f(x) \\ \text{when } x = w * p_n + r_n \text{ or } x = w * p_n + r'_n \text{ for } w \in \mathbb{N}.\end{array} \tag{2.13}$$

Rather than naively trying to divide each $f(x_i)$ by every element in $\{FB\}$ to check for smoothness as with Dixon, it is known if a particular $p$ divides a given $f(x_i)$ (although not how many times). This allows huge time savings since prime $p_n$ will be a factor in intervals of $p_n$ for consecutive $x_i$ in $f(x_i)$. In comparison with Dixon, this means half the divisions if $p = 2$, one-third the divisions for $p = 3$ and so forth throughout the factor base.

The factor base will also be smaller than with Dixon because not all primes less than $B$ will pass the Legendre test with respect to $N$. Looking back to Dixon's example, notice that $p = 5 \in \{FB\}$ never appears as a factor in any $f(x_i)$. Computing the Legendre symbol $(\frac{1633}{5}) = -1$, one can see 5 is a quadratic non-residue. Therefore 5 will *never* be a factor of *any* $f(x_i)$. Fewer primes in the factor base means shorter exponent vectors, less division, and faster matrix elimination.

Other than those modifications discussed here, QS runs identical to Dixon's method. One finds {FB}-smooth $f(x_i)$, stores relation pairs, and uses linear algebra to solve for a product of $f'(x_i)$ that create a perfect square.

## 2.5 Large Prime Variations

Consider the situation where an $f(x_i)$ is generated that is *near* {FB}-smooth. That is, $f(x_i)$ factors over the prime base except for a single prime, called the cofactor, $C$. Because $C$ divides an $f(x_i)$ its Legendre symbol $(\frac{N}{c}) = 1$. From this, it is known that $C > B$, otherwise $C$ would be in the factor base, and if $C < B^2$ then the Sieve of Eratosthenes guarantees us that $C$ is prime. Any $B < C < B^2$ is termed a large prime variant and the associated $(x_i, f(x_i))$ is called a partial relation. By combining partial relations that have the same large prime variant, it is possible to eliminate that variant and create a relation that is {FB}-smooth.

We will now show an example of two partial relations forming a smooth one. Consider factoring $N = 2701$ using smoothness bound $B = 13$. Under the new factor base restrictions of QS, this produces $\{FB\} = \{2, 3, 5, 13\}$. This setup will produce the two $f(x_i)$:

$$\begin{array}{rcrcll} f(62) & \Rightarrow & 62^2 & \equiv & 3^2 * 127 & \mod 2701 \\ f(65) & \Rightarrow & 65^2 & \equiv & 2^2 * 3 * 127 & \mod 2701 \end{array}$$

Combining these together, congruence holds across multiplication:

$$(62 * 65)^2 \equiv 2^2 * 3^3 * 127^2 \mod 2701$$

Finally, by multiplying both sides of the equation by $(127^{-1})^2 \mod 2701$, one can remove the large prime variant while maintaining congruence. Note: $127^{-1} \mod 2701 = 1191$.

$$\begin{array}{rcll} (1191 * 62 * 65)^2 & \equiv & 2^2 * 3^3 * 127^2 * 127^{-2} & \mod 2701 \\ 53^2 & \equiv & 2^2 * 3^3 & \mod 2701 \\ 53^2 - 2701 & \equiv & 2^2 * 3^3 & \mod 2701 \\ f(53) & \equiv & 2^2 * 3^3 & \mod 2701 \end{array}$$

Thus a valid {FB}-smooth relation pair has been produced and the algorithm can proceed as previously. It should be noted it requires two partial relations with the same $C$ to create

a single full relation, so the first such encounter is usually stored and used to process all subsequent occurences. Pomerance remarks, "the creation of these new exponent vectors is like a gift from heaven" [3].

Using large prime variations (LPV) has obvious time savings. It is possible to find relations using primes outside the factor base without increasing the size of the exponent vectors. This benefit comes with a time and memory overhead, though, since partial relations must be stored and retrieved. Variant primes near $B^2$ are unlikely to encounter a matching partial relation because of their size, so it is useless to store them. Instead, one should choose a large prime bound, $B_2$, on the interval $B_1 < B_2 < B_1^2$ (herafter, $B_1 = B$, the smoothness bound), and only consider $C < B_2$. We address the question of optimal sizing and relationship between $B_1$ and $B_2$ in Chapter 5.

# Chapter 3

# Sequential Algorithm

The sequential algorithm closely follows the mathematical description of the previous chapter. The relation acquisition portion of the algorithm will be discussed in great detail as it relates to the research question of Chapter 5. Matrix elimination, meanwhile, is necessary for QS but has little bearing on the topic of study.

## 3.1 Preliminaries

There are some well established optimization strategies so critical to an efficient implementation that they necessitate discussion prior to their individual details being encountered.

### 3.1.1 Block Strategy

With Dixon's method it was possible to generate a single $f(x_i)$, attempt to factor it over $\{FB\}$, then save $f(x_i)$ if it was a relation and otherwise discard it. Individual consideration of $f(x_i)$ is not optimal when using QS. Every $f(x_i)$ would have to check itself against the predicate of equation (2.13) for every prime in $\{FB\}$. Such a naive strategy returns QS to a runtime complexity near that of Dixon.

Instead, blocks of sequential $f(x_i)$ are used. Several thousand $f(x_i)$ or more are placed into an array. Then, for each $p \in \{FB\}$, the logic of equation (2.13) is used to find the

smallest $x_i$ in our array that satisfy the equality for $r$ and $r'$. These can be labeled $s$ and $s'$. As (2.13) demonstrates, handling the rest of the elements in the block is simple. If $p$ divides element $s$, it will divide $s + p$, $s + 2p$, ... and similarily with $s'$, $s' + p$, $s' + 2p$. ...

If (2.13) indicates that $f(x_i)$ is divisible by $p_x \in \{FB\}$ we remove all instances of $p_x$ in the factorization of $f(x_i)$ and set the new array value to that part which does not divide out. If quadratic residues indicate that a prime can be factored out of a $f(x_i)$, it is a guarantee of one or more such removals. In practice, divisions must be repeated until the prime does not divide evenly. This is done for the length of the block and all elements in $\{FB\}$.

The block is then traversed. The elements are now the cofactors of the corresponding $f(x_i)$. If the cofactor is 1, $f(x_i)$ is {FB}-smooth and the relation pair is stored; if the cofactor $< B_2$ then $(x_i, f(x_i))$ are suitable for partial handling. Otherwise, the pair is ignored. Blocks with increasingly higher values are sieved until sufficient relations are found.

### 3.1.2 Logarithmic Estimation of Factorization

Consider prime $p$, that via quadratic residues (2.13) is a known factor of $f(x_i)$. If $f(x_i)$ is large it is a costly operation to divide all instances of $p$ out of $f(x_i)$. Multiple divisions are always required and division is not efficient at the machine level. Futhermore, for large $N$, relations are rare among $f(x_i)$ and a significant percentage of runtime deals with sieving blocks. A large number of divisions are being performed, often on $f(x_i)$ which will never reduce to relations. This can be improved upon with the use of logarithms. Observe:

$$\text{If } Z = z * z' \text{ then } \log(Z) - \log(z) - \log(z') = 0. \tag{3.1}$$

The same logic is applied to numbers in the blocks. Rather than storing huge $f(x_i)$ values in memory, the comparatively tiny $\log(f(x_i))$ now populate the blocks. If (2.13) indicates a $p \in \{FB\}$ is a factor of a $f(x_i)$ the $\log(p)$ is subtracted from the block element corresponding to $f(x_i)$. As billions or more $f(x_i)$ are considered, the efficiency benefit of subtraction over division quickly becomes apparent.

Once a block has been completely sieved, it is traversed. Elements equal to 0 correspond to $f(x_i)$ which are $\{FB\}$-smooth and the product of unique primes. Being the product of unique primes is an unacceptably tight restraint. Observe that 2 completely factors 32, but $\log(32) - \log(2) \neq 0$. The logarithmic strategy is unable to compensate for instances where multiple removals of a single factor take place. So, instead of using 0, it will be checked to see if the subtractions reduce the block elements less than a predetermined error bound. For the basic QS, this is generally $\log(B_1)$ and $\log(B_2)$ is appropriate when large prime variations are used (these values are the smoothness bound and large prime bound, respectfully). Elements reduced less than the error bound are flagged as probable relations. The associated $f(x_i)$ are generated (in full) and undergo trial division over {FB} to determine the cofactor.

### 3.1.3 Technical Considerations

Prime factorization requires the use of large integer values beyond the primitives provided by today's hardware. The GNU Multiple Precision Library [4] (GMP) can be used for the storage and manipulation of large integers. The variable length integers are of type mpz_t at the code level. With over fifteen years of development and many common loops utilizing assembly language, GMP is a highly optimized mulitiple precision library. As such, GMP methods are generally preferred to manual implementations.[1]

## 3.2 Base Allocation and Population

We will start by jumping head first into the implementation details. Arrays are declared globally to make the psuedocode more concise. This also creates a convenient reference where they can be described.

```
ulong smoothArray[]   // Contains smooth primes w/Legendre == 1
ulong residueArray[]  // Contains lower quadratic residue for above primes
float logArray[]      // Contains log() of all smoothArray primes

ulong partArray[]     // Contains large primes w/Legendre == 1
```

[1] It should be assumed that any function name with prefix "mpz_" is part of the GMP library.

```
bool  hasPartial[]    // Indicates if large prime has stored partial rel.
mpz_t partX[]         // Contains x portion of partial rel.
mpz_t partFX[]        // Contains f(x) portion of partial rel.
mpz_t partROW[]       // Contains exponent vector portion of partial rel.

mpz_t relationX[]     // Stores x portion of smooth rel.
mpz_t relationFX[]    // Stores f(x) portion of smooth rel.
mpz_t expMatrix[]     // Matrix formed from exponent vectors of all smooth rels.

float curBlock[]      // Block currently being sieved/examined
```

The ulong user-defined datatype represents an unsigned long int which is used to store all primes. The main routine for the sequential Quadratic Sieve is provided below. The specific methods will be covered in detail in their individual sections.

```
quadSieveLPVSeq( mpz_t N, ulong smoothBound, float BRatio ) :
    ulong numSmoothP  // Number of (smooth) primes in smoothArray
    createSmoothArrays( smoothBound )
    numSmoothP = fillArrays( smoothBound, N )
    createLogArray( numSmoothP )
    ulong numPartP // Number of (large) primes in partArray
    ulong partBound = smoothBound * BRatio
    numPartP = createPartArray( N, smoothBound, partBound )
    createPartStructs( numPartP )
    createRelStructs( numSmoothP )
    findRelations( N, partBound, numSmoothP )
    augmentIndentity( expMatrix )
    matrixEliminate( expMatrix )
    processZeroRows( expMatrix )
```

The main routine requires three arguments. N is the number to be factored, the smoothness bound, and BRatio a variable used in the calculation of the large prime bound.

### 3.2.1 Factor and Residue Bases

Next, we must allocate memory for the factor and residue bases. Due to speed concerns, a dynamic approach is innappropriate. An array is sized using an overestimation of the prime counting function $\pi(x)$ so no primes are excluded. Implementations such as [1] have the user explicity enter the base size and then fill it to capacity with primes, resulting in no memory loss. Because the user has no idea how many primes will pass Legendre, though, he loses some element of control. Precise bound selection is necessary for our implementation since the analysis of Chapter 5 deals with optimizing that choice. Russian mathematician

Chebyshev observed:

$$\pi(x) < \frac{x}{\log_e(x) - 1.125} \tag{3.2}$$

Because these bases should never seriously challenge available memory, such an estimation is appropriate. With greater mathemetical complexity, use of the logarithmic integral yields a more accurate result. Primes lacking a quadratic residue and facing rejection from the factor base must still be compensated for. It has been proven that for prime moduli $p$:

$$\frac{p-1}{2} \text{ of integers are quadratic residues.} \tag{3.3}$$

In our case, moduli are always prime since they are potential factor base elements. Thus, 50% (probablistically) of the time N will produce a quadratic residue modulo $p$ and necessitate storage. The other 50% will be quadratic non-residues and are not stored. Using this logic, dividing Chebyshev's over-estimation by 2 should be a safe and somewhat accurate means to size smoothArray and residueArray. In cases where the smoothBound is small, however, the over-estimation Chebyshev provides will be minimal. Futhermore, a smaller number of primes being evaluated for residues causes greater deviation from the asymptotic 50% residue percentage. This causes memory faults when more factor base elements are required than estimated space allocated. This is remedied by increasing the above estimation by 10%, a strategy that eliminates this issue for reasonable smoothBound. A reasonable smoothBound is defined as one that would be used in the efficient factorization of a 25+ digit semiprime. Very small smoothness bounds may still fail under this method, but such bounds would only be effective in the factorization of trivially small semiprimes which are of no interest to QS. The creation of the arrays is shown below.

```
createSmoothArrays( ulong smoothBound ) :
    smoothEst = overestimatePrimes( 0, smoothBound )
    smoothArray = create array [ smoothEst ]
    residueArray = create array [ smoothEst ]
```

Since all primes must be less than the smoothness bound and all residues lie on the range of 1...(smoothBound-1), both bases will be of type ulong. With the arrays created they can now be initialized.

```
fillArrays( ulong smoothBound, mpz_t N ):
    smoothArray[0] = 2
```

```
residueArray[0] = 1
ulong numSmoothP = 1
ulong curPrime = 3
while curPrime <= smoothBound :
  if legendre(curPrime, N) == 1 :
    smoothArray[numSmoothP] = curPrime
    residueArray[numSmoothP] = findResidue( curPrime, N )
    numSmoothP++
  curPrime = nextPrime( curPrime )
return numSmoothP
```

Being the only even prime, 2 is problematic with some methods and is dealt with manually. On the assumption that N is an odd integer (else it is trivially factorable), 2 is known to be in the factor base with quadratic residue 1. We then proceed with the odd primes. Each prime less than smoothBound is passed along with N to mpz_legendre(·) to quickly determine residue existence. If a residue does exist, findResidue(·) is called.

```
findResidue( mpz_t N, ulong prime ) :
    for i = 1 to prime / 2 :
      if i^2 % prime == N % prime :
        return i
```

A brute force search on the interval $1 \ldots \frac{prime}{2}$ using mpz_congruent_p(·) is used to determine the residue solve value[2]. Prime incrementation is handled via mpz_nextprime(·) which uses trial division by small primes and then Miller-Rabin primality tests to locate the next prime. The number of prime/residue pairs is maintained in a variable named, numSmoothP.

### 3.2.2 Logarithm Base

As discussed in the preliminaries, logarithms are used to estimate the finding of smooth $f(x_i)$. During the sieving stage, the logarithms of smoothArray elements are frequently required. As such, it is insufficient to perform the division of the logarithmic operation each time it is needed during sieving. It is more appropriate to perform the computation once and store the values to memory with constant time access.

```
createLogArray( ulong numSmoothP ) :
    logArray = create array [numSmooth]
    for i = 0 to numSmooth - 1 :
      logArray[i] = log(smoothArray[i])
```

[2]Pomerance describes two residue solve methods in his text [3] which are much more efficient than brute force search. However, those are probablistic methods. Because runtimes are critical to our analysis a deterministic strategy must be used.

The estimation aspect of the logarithmic method justifies the use of floats instead of doubles. In fact, some practitioners advocate the use of integer accuracy logarithms to speed the sieving process. Our testing disputes these claims. Logartithmic round-off error leads to inaccurate probable relation determination. This causes true relations to be skipped or unnecessarily submits false relations to the expensive trial division verification.

### 3.2.3 Large Prime and Partial Relation Bases

As shown mathematically, two partial relations with the same large prime variation are needed to create a full relation; "the first such encounter is usually stored and used to process all subsequent occurences." [3] A collection of arrays, all with index correspondence are used to store partial relation data.

```
createPartArray( mpz_t N, ulong smoothBound, ulong partBound) :
    ulong partEst = overestimatePrimes( smoothBound+1, partBound )
    partArray = create array [partEst]
    while curPrime <= partBound :
      if legendre(curPrime, N) == 1 :
        partArray[numPartP] = curPrime
        numPartP++
      curPrime = nextPrime(curPrime)
    return numPartP

createPartStructs( ulong numPartP ) :
    hasPartial = create array [numPartP]
    partX = create array [numPartP]
    partFX = create array [numPartP]
    partROW = create array [numPartP]
```

Another modified Chebyshev estimation is used to gauge the number of primes on the interval (smoothBound...partBound] that have a Legendre symbol of 1. Then, the partArray is populated in much the same fashion as the smoothArray except that residues are not calculated or stored. After population, the precise number of large primes is known. This value, numPartP, is used to size several arrays whose names should be self explanatory. For example, if a partial relation with large prime variant $C$ is encountered we determine the index of $C$ in partArray via binary search. That position in hasPartial is then examined. If false, the composite parts of the current partial relation should be inserted into the other three structures. Otherwise, the current partial relation should be combined with the

existing partial relation data in the structures to form a complete relation that can then be handled like a smooth one.

## 3.3 Sieving and Matrix Construction

With the creation of the various structures complete, we can now begin the search for relations. It is most efficient to concurrently build the exponent vector matrix as relations are found.

### 3.3.1 Storing Relation Pairs

Before block sieving begins, data structures must be in place to hold relation pairs.

```
createRelStructs( ulong numSmoothP ) :
    ulong relCapacity = 10 + numSmoothP
    relationX = create array [relCapacity]
    relationFX = create array [relCapacity]
    expMatrix = create array [relCapacity]
```

We define $(10 + \mathsf{numSmoothP})$ relations to be a sufficient number. Why? Consider the case that only numSmoothP relations are required. This results in numSmoothP exponent vectors with numSmoothP columns, a square matrix prior to identity augmentation. This makes it possible for no linear dependencies to exist among the rows of the matrix and no zero rows produced when the matrix undergoes elimination.

Zero rows are a necessity for the algorithm. By requiring $(10 + \mathsf{numSmoothP})$ relations, linear algebra guarantees at least 10 zero rows. The probability of a zero row producing non-trivial factors was shown earlier to be 40% in the worst case. By this logic, the algorithm has

$$1 - (.60)^{10} \approx 0.9939 \tag{3.4}$$

or a 99.4% chance of finding the non-trivial factors.

This observation has significant effects on the runtime and memory complexity of the algorithm. Since smoothBound determines the size of numSmoothP which determines needed

relations, the user should pay close attention when choosing it. A large smoothBound will require many relations, but they will be easier to find since more $f(x_i)$ will be smoothArray-smooth. A smaller smoothness bound requires fewer, harder to find relations while decreasing the memory requirements for the exponent matrix. Also consider that a smaller exponent matrix makes for faster matrix reduction. Carl Pomerance, remarks, "The optimal $B$-value is more of an art than a science, and is perhaps best left to experimentation" [3]. The intricacies of this topic will be further studied in Chapter 5.

### 3.3.2 Storing Exponent Vectors

Since the exponent vector matrix is built concurrently while finding relations, a structure needs to be allocated for its storage. Essentially, the vector matrix is nothing more than a two-dimensional matrix of zeroes and ones. To avoid wasting memory, the underlying bit structure of a primitive type must be fully utilized.

It would not be difficult to write a module with bit-shifts and bitwise operators to manipulate a 2-D array of int type in such a way. But, realize that a variable length integer (mpz_t) is merely an efficient and organized means of performing this task with minimal overhead. The GMP library already has a reliable, optimized implementation with helpful methods and wraps this functionality into an easily manageable abstract object.

As a result, use the mpz_t type to store the lengthy exponent vectors. They are used as a data structure with no concern for the actual number the bits represent. The expMatrix array is allocated with the same dimension as the number of relations being searched for which was handled in psuedocode in the previous section. Individual rows are initialized as needed, which is when the vector (column) length is allocated.

### 3.3.3 Population of Sieving Blocks

The sieving process is the core of QS. We introduce a psuedocode method which outlines the process and facilitates discussion of the individal components.

```
findRelations( mpz_t N, ulong partBound, ulong numSmoothP ) :
    mpz_t blockStart = sqrt(N)
```

```
mpz_t upperBound = sqrt(2N)
ulong relsFound = 0
curBlock = create array [BLOCKSIZE]
while blockStart <= upperBound && relsFound != (10+numSmoothP) :
  populateBlock( blockStart, BLOCKSIZE )
  sieveBlock( blockStart, BLOCKSIZE )
  testProbs( &relsFound, partBound, numSmoothP )
  blockStart += BLOCKSIZE
```

Before entering the sieving loop, we initialize several variables. The lower and upper limits on sieving are calculated and the sieving block is allocated. The block has BLOCKSIZE elements. Because it is desireable to tweak BLOCKSIZE based on the size of N being factored, one may think it appropriate to read it from the command-line or solve for the value inside the code. Instead, define BLOCKSIZE as a pre-processor directive and recompile each time the value changes. BLOCKSIZE serves as the conditional in many loops during the iterative sieving process and if the compiler knows this value it can make *significant* runtime improvements. Runtime reductions up to 20% were observed in our tests.

The sieving loop is then entered, a loop only broken if upperBound on $x$ is reached or sufficient relations are found. The first step is to populate the block with $\log(f(x))$, as discussed in the preliminaries. To begin, x is passed to f(x) = $x^2-N$. The f($\cdot$) are so large they must be stored temporarily as mpz_t. But, this huge value is of no interest only its logarithm is. The GMP library, however, has no function to return the logarithm of an integer. The GMP has a sister library, the MPFR [2], specializing in floating point math that has logarithmic functions, but the extra time needed to cast over each f($\cdot$) and perform costly floating point divisions makes such a choice unacceptable.

Instead, the logarithm of each f($\cdot$) is estimated using a rather crude method. The function mpz_sizeinbase(mpz_t num, int z) returns the number of digits that appear in the base-z representation of num. Observe that the $\log_{10}(100) = 2.0$, and mpz_sizeinbase(100, 10)$-1 =$ 2. At the other end of the estimation spectrum $\log_{10}(999) = 2.9995$ but mpz_sizeinbase(999, 10)$-1 = 2$, almost an entire integer away from the correct answer. This is a large margin of error. It is important to remember, though, that logarithmic *estimation* is being performed, and the error bound against which probable relations are checked will compensate for these

indiscretions. To reduce the estimation error log base 2 will be used instead of base 10.

Another estimation trick can improve runtimes even more. Every call to $f(\cdot)$ requires a GMP squaring and subtraction operation. These values are precisely calculated, only to have this precision destroyed milliseconds later when a crude logarithmic estimation is calculated. As the values of x passed to $f(\cdot)$ become large there could be thousands of consecutive elements which, even after precise $f(\cdot)$ calculation, have the same integer-accuracy log value. Therefore, why waste time performing precise calculations? Instead, we estimate the expected integer logarithm value for a block and set every element in the block to that value. The following psuedocode shows how block population *should* be performed:

```
populateBlock( mpz_t blockStart, ulong BLOCKSIZE ) :
    for i = 0 to BLOCKSIZE - 1 :
      curBlock[i] = numBitsIn(f(blockStart+(BLOCKSIZE/2)), N)

f( mpz_t x, mpz_t N ) :
    return x^2-N
```

In order to produce a single $f(\cdot)$ that is a good representative of an entire block, you should consider the median element. Calculate f(blockStart+(BLOCKSIZE/2)), pass it to the logarithm estimation, and use this value to populate all block elements. Indeed, this takes the estimation psuedo-science to a new level. While deficencies to the strategy exist, they do not outweigh what amounts to constant-time block population.

The only estimated block element known equal to its previously non-estimated counterpart is the median element. Remember that the goal of sieving is to reduce these elements less than some error bound. However, the error bound is sufficiently sized to deal with minor over/under-estimations of just a few integers (errorBound = $log_2$(partBound)). If it can be shown that the range of a fully calculated block would only be several integers, this strategy would be beneficial.

Recall that $f(x) = x^2 - N$. Because N is a constant, attention should be focused on the $x^2$ term. Additonally, the logarithm estimation solely counts the number of bits in the binary representation of f(x). As a result, Sloane's OEIS [9] entry A126726[3] is of great interest.

[3]Sloane's A126726 - The number of squares (of nonnegative integers) that require n binary (base 2) digits: {0, 2, 0, 1, 1, 2, 2, 4, 4, 7, 9, 14, 18, 27, 37, 54, 74, 107, 149, 213, 299, 425, 599, 849, 1199, 1697, 2399, 3394, 4798, 6787, 9597, 13573, 19195, 27146, 38390, 54292, 76780, 108584, 153560, 217168, 307120, 434335, 614241, 868669, 1228483, 1737338, 2456966}.

This sequence gives an idea of how fast bit expansion, and thus $\log_2$ integer growth, occurs in f(·). While the sequence is very volatile at the outset, it begins to validate our strategy as thousands and even millions of consecutive squares require the same number of binary digits. The expotential sequence growth indicates that the population strategy will become increasingly accurate as more blocks are sieved, making the strategy especially appropriate for very large N requiring such extensive searching.

This has implications when choosing BLOCKSIZE. On the one hand, it is desireable for BLOCKSIZE to be large because this reduces the number of blocks to sieve and the overhead associated with each. On the other, a large block size increases the range of mis-estimation. An appropriate compromise must be made between these factors, usually resulting in a block of between $50,000$ and $150,000$ elements depending on the size of N.

Finally, the first block(s) are the most poorly estimated because f(x) growth has not yet reached logartithmic consistency. Unforuntely, these first blocks are the most likely to produce relations since they generate small f(·) which are more likely to be smoothArray-smooth. Strategies were experimented with where only the first (or first several) blocks were precisely calculated, and the rest use the estimation strategy. The time wasted making precise calculations even for these small numbers outweighs estimation benefits.

### 3.3.4 Prime Removal (Sieving)

The next step is to subtract logArray elements from block elements in accordance with the residue solve values. At the heart of this step is the blkEntry(·) method that returns the first index in the current block for which prime is a factor of the corresponding f(·). Pseudocode demonstrates how the process is performed:

```
sieveBlock( mpz_t blockStart, ulong BLOCKSIZE ) :
    for i = blkEntry( blockStart, 2, 1 )  to BLOCKSIZE - 1 by 2 :
      curBlock[i] -= logArray[0]
    for i = 1 to numSmoothP - 1 :
      ulong prime = smoothArray[i]
      ulong residue = residueArray[i]
      for j  = blkEntry( blockStart, prime, residue ) to BLOCKSIZE - 1 by prime:
        curBlock[j] -= logArray[i]
      residue = prime - residue
      for j  = blkEntry( blockStart, prime, residue ) to BLOCKSIZE - 1 by prime:
```

```
        curBlock[j] -= logArray[i]

blkEntry( mpz_t blockStart, ulong prime, ulong residue ) :
    ulong startPos = -1 * (blockStart/prime) + residue + prime
    if startPos > prime :
      starPos -= prime
    return startPos
```

Remember that residues exist in pairs and that we must sieve using both of them, despite the fact only one is stored. In addition, we know that 2 is in the prime base with quadratic residue 1. Calculating the symmetric (second) residue $2 - 1 = 1$ evaluates to the same value. If 2 is handled like other primes, every time 2 appears as a factor in an $f(\cdot)$, two such instances will be subtracted out whether the second exists or not. Prime 2 is handled as a separate case so only the first residue is used.

### 3.3.5 Relation Search and Verification

Once the block has been sieved probable relations need to be identified and processed.

```
testProbs( ulong relsFound, ulong partBound, ulong numSmoothP ) :
    mpz_t expRow
    float errorBound = log(partBound)
    for i = 0 to BLOCKSIZE - 1 :
      if curBlock[i] < errorBound :
        ulong cofactor = relVerify( f(blockStart+i), numSmoothP, expRow)
        if cofactor == 1 :
          relationX[relsFound] = blockStart + i
          relationFX[relsFound] = f(blockStart + i)
          expMatrix[relsFound] = expRow
          relsFound++
        else if cofactor < partBound :
          ulong index = findCofactor( cofactor ) // Binary search in partArray
          if hasPartial[index] :
              // See section 2.5 on combining partials to make smooth rels.
            relationX[relsFound] = makeSmoothX(blockStart+i, partX[index])
            relationFX[relsFound] = makeSmoothFX(f(blockStart+i), partFX[index])
            expMatrix[relsFound] = expRow XOR partROW[index]
            relsFound++
          else :
            partX[index] = blockStart + i
            partFX[index] = f(blockStart + i)
            parROW[index] = expRow
            hasPartial[index] = true
```

Each element of the block is visited and checked against the error bound. If a block element is less than errorBound, the element is flagged as a probable relation and the corresponding

f(x) is generated in full. An additional mpz_t named expRow is allocated with numSmoothP bits using the mpz_init2(·) function. If the probable relation verifies to be a smooth or partial relation, expRow is its mod 2 exponent vector. If it does not, the vector is ignored and the memory deallocated.

The function relVerify(·) has the dual purpose of trying to factor f(·) over the factor base and concurrently building the exponent vector.

```
relVerify( mpz_t probRel, ulong numSmoothP, mpz_t &expRow ) :
    for i = 0 to numSmoothP - 1 :
      if(( mpz_remove(probRel, probRel, smoothArray[i]) % 2 ) == 1 ) :
        mpz_setbit( expRow, i )
    return probRel
```

The method mpz_remove(·) is used to perform the individual divisions. This function removes all occurences of a factor (the third argument) from the probRel. The result is stored to the first argument and the return value is how many such removals took place. For our purposes, the factor will always be a ulong from the factor base. Casting every smoothArray element to mpz_t every time relVerify(·) is called is unnacceptable. Opening the source to the GMP package, a modification allows a ulong to be passed to the function without the speed reduction of a type cast[4].

A traversal across the factor base is done with the result of one call becoming the op of the next. Intialization of expRow sets all the bits to 0, so only if the return value of mpz_remove(·)%2 = 1 does mpz_setbit(·) need to be used to set the appropriate exponent vector column. After the factor base has been traversed, probRel is the cofactor, and this is the value returned by function relVerify(·).

As covered mathemetically, if the cofactor is 1 a smooth relation has been encountered and the relation parts may be stored. If the cofactor is less than partBound, the cofactor is a large prime variation, and the data can be handled in accordance with the explanation of section 3.2.3. Otherwise, the probable relation is ignored. Note that while the expMatrix size and type have been declared, individual rows are intialized on an as-needed basis. Using

[4]Basically, the GMP function sets the argument f to an array position using mpz_set(·). In the modified version, the same can be done for an argument ulong by using mpz_set_ui(·)

mpz_init2(·) each row is given (2∗numSmoothP+10) bits to accomodate the exponent vector and space for the identity.

Building expRow for probable relations and simply deallocating it if they do not verify is advantageous. The setting of bits is a constant-time operation. The cost lies in the divisions which determine the bit-sets. Time lost bit-setting relations that do not verify is minimal compared to re-dividing verified relations to find bit representations.

## 3.4 Matrix Elimination

Post-sieving, the diagonal bits are set along the identity and the matrix is ready for elimination. Though Gaussian elimination is generally considered to be a "basic" solve method, its implementation in $\mathbb{F}_2$ is simple, yet surprisingly efficient for moderately sized matrices. As already demonstrated, and Pomerance [3] notes, a matrix over $\mathbb{F}_2$ "naturally lends itself to computer implementation."

Gaussian elimination uses only elementary row operations (swaps, scaling, subtraction) all of which are simplified in $\mathbb{F}_2$ compared to their decimal forms. Row swaps are achieved by exchanging pointers, something mpz_swap(·) handles quite nicely. Reduced row echelon form requires all pivots be 1 and non-zero entries in the pivot column must equal the pivot before subtraction can take place. Scaling is used to achieve these characteristics, an operation no longer required because there is only one non-zero array entry. Lastly, row subtraction in $\mathbb{F}_2$ reduces to bitwise XOR, a quick machine level operation

Our research does not concern itself with work past the sieving phase, so matrix elimination is a non-factor. Our source code does include a basic Gaussian elimination implementation, moreso to check algorithm correctness than to actually be useful in the factoring of numbers. When matrices grow large and elimination becomes time critical, more complex techniques are appropriate.

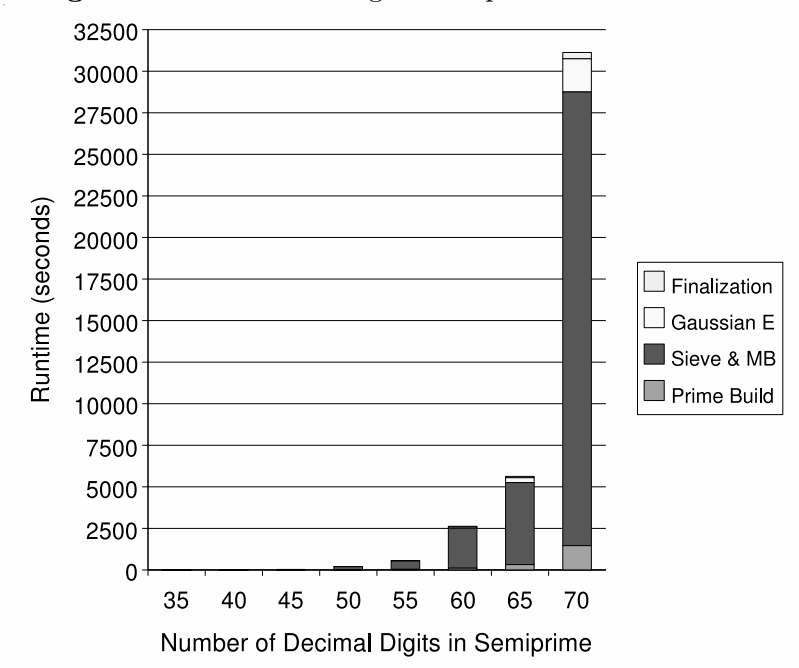
Algorithms such as Block Lanczos, Conjugate Gradient, and Wiedmann's have all been brought to bear on this problem. Their mathematical foundations are too complex to

discuss here, but it suffices to say these methods specialize in the solving of sparse linear systems. In fact, the number on non-zero entries in the matrix determines runtime, not the size of the matrix itself. These advanced techniques can manipulate matrices in compressed forms, such as list representations, which alleviate memory requirements.

Once solution vectors have been determined, the congruences of squares can be built up as equation (2.7) describes. After running the a, b pairs through mpz_gcd(·), trivial and repititous solutions are discarded and the factors of N can be dealt with as the implementor sees fit.

To conclude the chapter, we present graph Figure 3.1 which gives some idea of the runtimes for individual sections of the algorithm, and the algorithm as a whole.

**Figure 3.1**: Phase Timings for Sequential Factorizations



# Chapter 4

# Parallel Implementation

Parallelization of the Quadratic Sieve (QS) is a necessity to sieve and factor numbers of a significant magnitude. The sequential factorization of $T_{70}$ takes about 8 hours, but when 32 nodes are brought to bear on the problem it completes in 19 minutes. Fortunately, the relation acquisition portion of the algorithm on which our work is concerned is easily parallelizable. Attention will be given to the parallel strategy in general and a few of the complications typically encountered. Parallel counterparts to the matrix elimination methods of Chapter 3 exist and the current literature does them more justice than can be given here.

The parallel method determined most efficient for QS uses the master-slave organization which allows a master node to facilitate storage while compute nodes handle the majority of the computational workload. Such a strategy leaves the master node underloaded the majority of time, but this keeps communications pipelines free and ensures that all compute nodes run at maximum efficiency. This also permits the efficient use of simple synchronized blocking communications. The Message Passing Interface (MPI) was used for the implementation.

## 4.1 Communicating Multiple Precision Integers

mpz_t integers, not being a primitive datatype, cannot be simplistically sent and received in MPI. The solution is to first declare a character array to act as a buffer. The mpz_t is then converted to a c-style string using mpz_get_str(·) which is inserted into the buffer. The first strlen(·)+1 characters of the buffer can then be communicated using the primitive MPI_CHAR type. At the other end, an equally sized buffer receives the string which is converted back to a GMP integer using mpz_set_str(·). Buffer sizing varies based on circumstances.

The radix of these conversions is also important. To minimize the number of characters sent, it is ideal to use a large base. GMP documentation details that string to mpz_t conversion is possible in bases 2 to 62, but the reverse is limited at bases 2 to 36. Because both directions must be utilized, 36 is the largest possible base. Since 32 is a power of 2 it is quick to convert to binary and vice versa, justifying the use of base 32 for all character based mpz_t communications.

## 4.2 Factor Base Construction

Just as in the sequential psuedocode all arrays are declared globally, and the basic outline of the parallel version is described below. Some methods are re-used from the sequential version and time will not be taken to re-describe them.

```
quadSieveLPVPar( mpz_t N, ulong smoothBound, float BRatio ) :
    if nodeNum == 0 :
      rootNode( N, smoothBound, BRatio )
    else :
      computeNode()

rootNode( mpz_t N, ulong smoothBound, float BRatio ) :
    broadcast( N )
    ulong partBound = smoothBound * BRatio
    broadcast( partBound )
    createSmoothArrays( smoothBound )
    numSmoothP = fillArraysRoot( smoothBound, N )
    createLogArray( numSmoothP )
    bcastStructs( numSmooth P )
    ulong partBound = smoothBound * BRatio
```

```
    ulong numPartP = createPartArray( N, smoothBound, partBound )
    createPartStructs( numPartP )
    createRelStructs( numSmoothP )
    findRelationsRoot( N, partBound, numSmoothP )
    augmentIndentity( expMatrix )
    matrixEliminate( expMatrix )
    processZeroRows( expMatrix )
    cleanupMessages()

computeNode() :
    broadcast( mpz_t N )
    broadcast( ulong partBound )
    fillArraysCompute( N )
    broadcast( ulong numSmoothP )
    recvSmoothArrays( numSmoothP )
    findRelationsCompute( N, partBound, numSmoothP )
```

The first real change from the sequential version deals with constructing the factor base.

```
fillArraysRoot( ulong smoothBound, mpz_t N ) :
    smoothArray[0] = 2
    residueArray[0] = 1
    ulong numSmoothP = 1
    ulong curPrime = 2
    int quitNodes = 0
    while quitNodes != numProcs-1 :
      curPrime = nextPrime( curPrime )
      while legendre(curPrime, N) != 1 :
        curPrime = nextPrime( curPrime )
      receive( ulong prime, ulong residue )
      if prime != 0 :
        smoothArray[numSmoothP] = prime
        residueArray[numSmoothP] = residue
        numSmoothP++
      if curPrime <= smoothBound :
        send( curPrime )  // Send to node who just returned residue
      else :
        send( quit )
        quitNodes++
    return numSmoothP

fillArraysCompute( mpz_t N ) :
    send( 0 ) // Request data from root node
    while message.tag != quit:
      receive( ulong prime )
      send( findResidue( prime, N ) )
```

The master node sequentially dispatches odd primes with a Legendre symbol of 1 (as previous, 2 is handled manually) to available compute nodes that use brute-force search to find and return the residue solve values. Once all primes less then or equal to smoothBound have

been handled a quit message is sent to break the send-receive loop at the compute level. Once all slaves have been sent a quit message, the root node can proceed.

The principle package of communication during this process is an array of two ulongs. The first element contains the prime and the second its residue. Grouping the two into a single array prevents race conditions that could arise from using multiple send statements or making runtime assumptions. It is unnecessary to compute log2(smoothArray) in parallel because it is a sufficiently quick operation whose runtime is less than that of communications overhead.

When this phase is complete, the bases contain the same data as in the sequential version. However, the data is not necessarily ordered the same because some quadratic residue solutions are found quicker than others. Because all three bases (smooth, residue, and log) still have index correspondence, this does not affect the algorithm.

```
bcastStructs( ulong numSmoothP ) :
    broadcast( numSmoothP )
    broadcast( smoothArray )
    broadcast( residueArray )
    broadcast( logArray )

recvSmoothArrays( ulong numSmoothP ) :
    smoothArray = create array [ numSmoothP ]
    residueArray = create array [ numSmoothP ]
    logArray = create array [ numSmoothP ]
    broadcast( smoothArray )
    broadcast( residueArray )
    broadcast( logArray )
```

While the master node stores this base data, it is in fact the only node that does not need it. Only relation searching and verification requires the bases, a task performed exclusively on compute nodes. They must be broadcast so all compute nodes have a copy. Prior to broadcasting these structures, numSmoothP can be dessiminated allowing precise allocation. The memory of compute nodes does not have to suffer from Chebyshev's overestimation.

Since this data does not need to be retained on the master node it can be deallocated there. Due to the tree like structure in which MPI_Bcast(·) dessiminates data, it is unsafe to immediately deallocate the memory. Broadcasting is not a completely blocked communication method. Immediate deletion causes signal errors to be thrown. To remedy this, an

MPI_Barrier(·) is placed just prior to deallocation on the master and after successful base reception on each compute node. The barrier ensures a fully blocked broadcast.

It should be noted the remarks of this section apply only to the factor, residue, and logarithm bases. The relation storage structures, including the many neccesary to store partial relations, are sized as sequentially and exist solely on the master node. Their role in parallel interactions will be covered in the next section.

## 4.3 Sieving and Matrix Construction

The basic strategy is for compute nodes to locate relations and compute associated exponent vector rows. These will be "packed" into a single buffer that will be sent to the root for storage. Blocks are processed until sufficient relations are found or all compute nodes reach the upper bound on sieving. Psuedocode describes the master node's role in the process:

```
findRelationsRoot( mpz_t N, ulong partBound, ulong numSmoothP ) :
    ulong relsFound = 0
    int quitNodes = 0
    while relsFound != numSmoothP+10 && quitNodes != numProcs-1 :
      receive( message )
      if message.tag == quit :
        quitNodes++
      if message.tag == smooth :
        unpack( mpz_t x, mpz_t row from message )
        relationX[relsFound] = x
        relationFX[relsFound] = f( x )
        expMatrix[relsFound] = row
        relsFound++
      else :
        unpack( ulong cofactor, mpz_t x, mpz_t row from message )
        ulong index = findCofactor ( cofactor )
        if hasPartial[index] :
            // See section 2.5 on combining partials to make smooth rels.
          relationX[relsFound] = makeSmoothX( x, partX[index] )
          relationFX[relsFound] = makeSmoothFX( f(x), partFX[index] )
          expMatrix[relsFound] = row XOR partROW[index]
          relsFound++
        else :
          partX[index] = x
          partFX[index] = f( x )
          parROW[index] = row
          hasPartial[index] = true
```

At the compute level, the basic strategy is for compute nodes to sieve every $(numProcs - 1)$

block. Local variable nodeNum determines which block each node is responsible for in that interval. Once a compute node has a block, it handles it as done sequentially, except for sending all relations to the root for storage or partial processing.

```
findRelationsCompute( mpz_t N, ulong partBound, ulong numSmoothP ) :
    mpz_t blockStart = sqrt(N) + BLOCKSIZE * nodeNum
    mpz_t upperBound = sqrt(2N)
    curBlock = create array [BLOCKSIZE]
    while blockStart <= upperBound :
      populateBlock( blockStart, BLOCKSIZE )
      sieveBlock( blockStart, BLOCKSIZE )
      testProbsCompute( partBound, numSmoothP )
      iprobe( bool isMessage)
      if isMessage :
        quit
      blockStart += BLOCKSIZE * numProcs
    send( quit )

testProbsCompute( ulong partBound, ulong numSmoothP ) :
    mpz_t expRow
    float errorBound = log(partBound)
    for i = 0 to BLOCKSIZE - 1 :
      if curBlock[i] < errorBound :
        ulong cofactor = relVerify( f(blockStart+i), numSmoothP, expRow)
        if cofactor == 1 :
          pack( blockStart+i, expRow into buffer)
          send( buffer )
        else if cofactor < partBound :
          pack( cofactor, blockStart+i, expRow into buffer )
          send( buffer )
```

Indeed, a more even block distribution can be achieved by allowing the root node to distribute blockStart values on an as-needed basis using a workpool strategy. Blocks that contain relations take much longer to process than relation-free ones. With the discussed block population and logarithmic estimation improvements, however, blocks can be sieved very quickly. Communications needed to distribute blockStart values are so frequent that the master becomes backlogged with messages, causing idling across the virtual machine. The workload division of the previous paragraph proves more efficient because it can be handled locally.

Slave nodes proceed by populating, sieving, and verifying relations in the current block. Only if a probable relation verifies (partial or complete) must communication take place. Just as a small array is utilized in prime-base building, MPI_Pack($\cdot$) is used to combine

associated data elements into a single buffer to prevent race conditions. Packing variable length mpz_t string representations into a buffer is more complicated than communicating a statically sized array of primitives. In addition to the data itself, information must be sent about the length of each data element to allow successful unpacking on the received end. Fortunately, MPI_Pack(·) was designed with this purpose in mind.

If a smooth relation is found, four items are packed into the buffer: the length of x, x itself, the length of the exponent row, and the exponent row itself. Partial relations pack an additional element, the cofactor. These messages are distinguished using tags msg_SMOOTH and msg_PARTIAL so the master can unpack them correctly. Notice that the f(x) is never sent. By sending x, the master can pass this value off to f(·) to generate the other half of the pair. This proves quicker than converting f(x) to a string, packing it, communicating the extra bytes, unpacking, and converting back to an integer.

Sizing the relation buffer is more complex than when transmitting single values, like N. This buffer is substantial in size and, to avoid wasting memory, is precisely sized as:

$$\mathsf{BUFFSIZE} = \left( \frac{\mathsf{numSmoothP} + \mathsf{mpz\_sizeinbase(upperBound,\ 2)}}{5} \right) + 16. \qquad (4.1)$$

In the numerator, the number of bits in the exponent row and largest possible x (end of interval) are summed. This is divided by 5 since communications happen in base 32, not binary. Space for the cofactor (4 bytes, a ulong), two pack length integers ($2 * 4$ bytes), two null terminators ($2 * 1$ bytes), and compensation for roundoff error ($2 * 1$ bytes) explain why an additional 20 bytes are allocated.

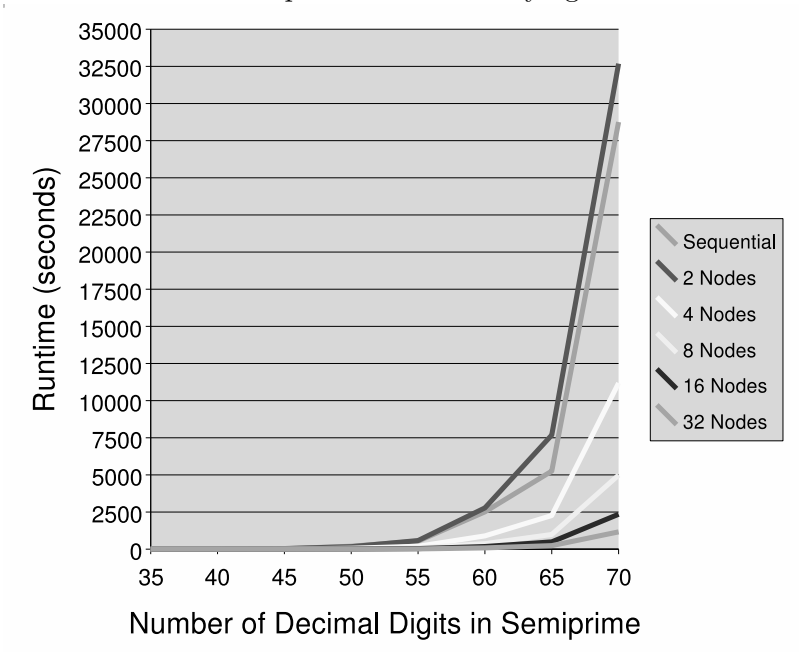
With the sieving data divided between master and compute nodes it is more complicated to handle termination conditions. Compute nodes are capable of checking locally to see if they are about to sieve a block whose blockStart $>$ upperBound. If this is the case, the slave will send a quit message to the master who will mark the node as dead. The master stays in its send-receive loop until all slaves are marked dead (and thus all blocks sieved) or a sufficient number of relations are found. If either condition is met, the master proceeds *immediately* with its linear algebra phase, which in our case, is done serially. Having not been informed of this transition, active slave nodes will continue sieving while matrix

elimination takes place. Outstanding messages are collected at the algorithms conclusion and each node is sent shutdown instructions.

```
cleanupMessages() :
    while quitNodes != numProcs-1 :
      receive( message )
      if message.tag != quit :
        send( quit ) // Send to node who we just received from
      else :
        quitNodes++
```

Slaves call MPI_Iprobe(·) after each block they sieve in anticipation of this fact, and if a message exists, the sieving loop is broken. Confirmation of successful shutdown is sent and once all slave nodes have been marked as dead the virtual machine can finalize and exit. As graph Figure 4.1 below shows, once one compensates for the fact the master node does not participate in sieving, the implementation shows near ideal speedup percentages.

**Figure 4.1**: Runtime Comparison Across Varying Size Virtual Machines



# Chapter 5

# Bound Optimization

Relatively accurate prime bound selection is necessary for the Quadratic Sieve (QS) algorithm to operate efficiently. Recall that prime bounds are the upper limits on the factor and large prime bases; $B_1$, $B_2$ as they were referred to mathematically or smoothBound, partBound in the implementation discussion. In this chapter, attention focuses on choosing bounds such that the requisite number of relations are found in a minimal amount of time.

Also remember that the selection of $B_1$ determines the number of relations one must find. A large $B_1$ will require many, easy to find relations. A smaller choice needs fewer, harder to find relations. Striking a compromise between these two facts is the first order of business. This will be done using basic QS, a stripped-down version without large prime variations (LPVs), and thus no $B_2$. The information from these simplified trials will be helpful when LPVs are then introduced to the analysis. Number theory will be useful in approximating bound selection. Runtime data will help refine this basis. Additionally, it will allow bound consideration from a less theoretical point of view. For example, approximation strategies might alter runtimes in ways number theory can not anticipate. It is the ultimate goal to produce a function, that given $N$, will output reasonable bounds.

We must realize since this analysis attempts to minimize sieving time, it is ignorant of matrix elimimination. Elimination is not only a very significant portion of QS but is also intimately tied to bound selection. Increasing $B_1$ makes exponent vectors longer and results

in polynomial growth of the matrix as a whole. When partial relations (those with cofactors on the range $B_1 \ldots B_2$) are combined to produce a full relation the resulting exponent vector is usually more dense than those from completely smooth relations. Optimal bound selection for the algorithm as a whole must take these factors into account. However, analysis of the linear algebra aspects of QS will be left to those with more expertise in the subject. Ideally, such a person could create a function detailing how bound growth affects a particular matrix elimination method's runtime. By summing that function with our own one could begin determining ideal bounds for QS. For our purposes, though, elimination concerns will be ignored and minimal mention of them will be made beyond this point.

## 5.1 Basic Quadratic Sieve

Pomerance's text [3] establishes most of the theory necessary for basic QS. Heuristically, Pomerance has shown that the probability of a single $x$ being $B$-smooth is

$$Prob(\ Smooth\ ) = u^{-u} \text{ where } u = \frac{\ln N}{2 \ln B}. \tag{5.1}$$

Using this, one can deduce that on the average, $u^u$ $f(x_i)$ must be considered to find a single relation. Combining this with some assumptions about the number of relations to be found and taking the derivative to obtain the absolute minimum of the function, Pomerance concludes the optimum B-selection to be

$$B_1 = e^{\frac{1}{2}\sqrt{\ln N \ln \ln N}}. \tag{5.2}$$

To see how our implementation matches up to these probablistic claims, we execute it using the suggested $B_1$ value and compare Pomerance's $u^u$ to runtime statistics. As Table 5.1 shows, the number of $f(x_i)$ that must be considered to find a relation is about 400 times greater than theory suggests for our test numbers. Assuming the accuracy of Pomerance's construction, this means only a small fraction of relations are actually being found under estimation strategies. While an astonishing statistic one can not deny that these estimations are beneficial. While $T_{40}$ can be factored sequentially in under 10 seconds the same

calculation without logarithms and constant time block population takes hours. Clearly though, Pomerance's optimal $B_1$-value is not applicable to the implementation.

**Table 5.1**: Relation Occurence Rate, Theory vs. Implementation

| $N$ | $B$ | $u^u$ | Rel. Interval |
|---|---|---|---|
| $T_{35}$ | $11,541$ | $492$ | $325,792$ |
| $T_{40}$ | $25,846$ | $870$ | $234,815$ |
| $T_{45}$ | $55,353$ | $1,492$ | $487,828$ |
| $T_{50}$ | $114,278$ | $2,498$ | $933,235$ |
| $T_{55}$ | $229,028$ | $4,103$ | $1,165,120$ |
| $T_{60}$ | $446,145$ | $6,613$ | $4,158,953$ |
| $T_{65}$ | $849,497$ | $10,499$ | $3,130,561$ |

With more distance between relations than anticipated, one may immediately assume that a much larger $B_1$ is appropriate. Let us pause to examine some reasons why a humongous $B_1$ is not the best selection. First, factor and residue base construction. All primes less than $B_1$ must be generated, passed to Legendre, and have their residue values solved. The latter is done using a brute-force method with expotential time complexity. If $B_1$ is very large this could be a significant bottleneck.

Secondly, memory has been given little attention to this point, but it is a factor. Paging to virtual memory is an unnacceptable slowdown for the most frequently needed structures (log, residue, and smooth bases). Attention should be given to ensure that these structures reside entirely in physical memory. A very large $B_1$ could strain memory availability on older systems since it is involved in sizing nearly every data structure.

Third, recall that sieving (fast) is used to flag probable relations that undergo verification via trial division (slow) over the factor base. A large $B_1$ means a larger factor base and more costly divisions in each verification. With multiple precision integer involvement, division can no longer be considered a constant time operation. Additionally, with larger $B_1$ the verification process as a whole will be called more often, since more relations are needed.

Fourth, in the sequential version once relations are found they are just simply inserted into the appropriate arrays. Matters are not so simple in the parallel case. Relation el-

ements must be converted to strings, packed into a buffer, communicated, unpacked, and de-converted. For this reason, it may be undesireable to have too many relations because of local slow downs. In addition, if $B_1$ is very large, relations will be found and communicated frequently. The onslaught of messages to the master may be more than it is capable of handling, a large queue may develop, and the send instructions will become blocked. A large number of nodes in the virtual machine would exacerbate this effect.

In contrast, one additional fact supports the need for a large $B_1$. Do not forget that $\log_2(B_1)$ serves as the error bound when logarithmically sieving blocks. If $B_1$ is small the factor base will be as well. For an $f(x_i)$ to be smooth over a set of small primes there will need to be multiple instances of those primes. Because the implementation does not re-sieve with higher powers of primes this is not compensated for. The error bound is in place to provide lee-way for this fact, but if $B_1$ selection dictates a small error bound then the situation is dire. $F(x_i)$ which are truly relations will not be flagged as such by estimation, and $f(x_i)$ growth will only increase the probability that multiple-prime instances occur as the method proceeds.

Trying to mathematically quantify the affect these factors would have on runtime would be tiresome and probably inaccurate. Instead, runtimes are calculated, incrementing and decrementing the smoothness bound via iterative methods until one arrives at a reasonably optimum solution. Because of the quantity of test runs it is necessary to use small/moderately sized semiprimes. Processing data from these small runs will hopefully allow us to make some assumptions about larger integers. Pomerance's $B_1$ construction will initialize the process.

Before trying to use runtimes to work backwards towards optimum bounds one needs to understand algorithm behavior. QS is a Las Vegas algorithm, meaning that runtimes are not completely deterministic. If Legendre rejects many small primes from the factor base the algorithm will have a harder time finding relations, leading to runtime variance. The differences will never be so dramatic as to void the usage of runtimes, but it should be noted that no regression can be perfect and some range of error is therefore expected in all

calculations. With this in mind, runtime analysis may continue.

**Figure 5.1**: $T_{65}$ Runtimes Using Basic QS w/Varying Bounds

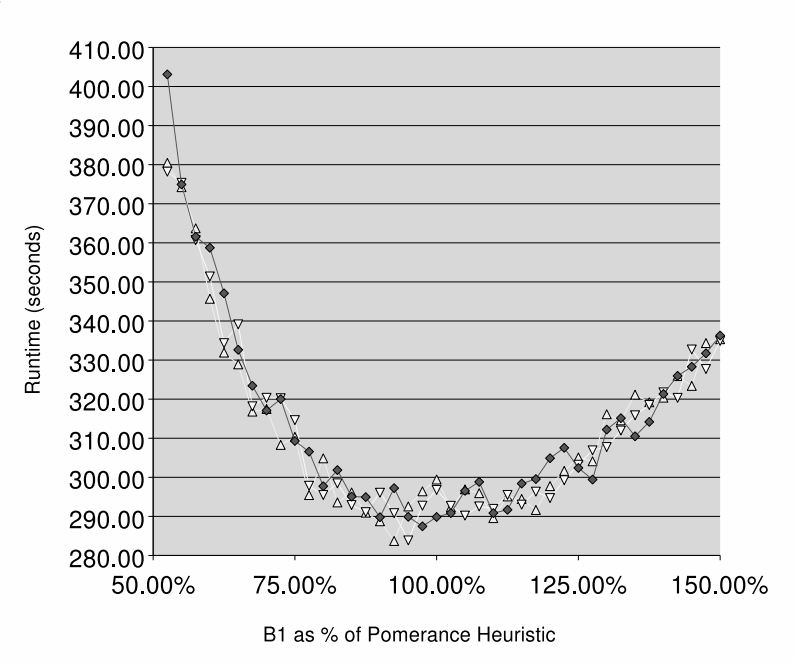


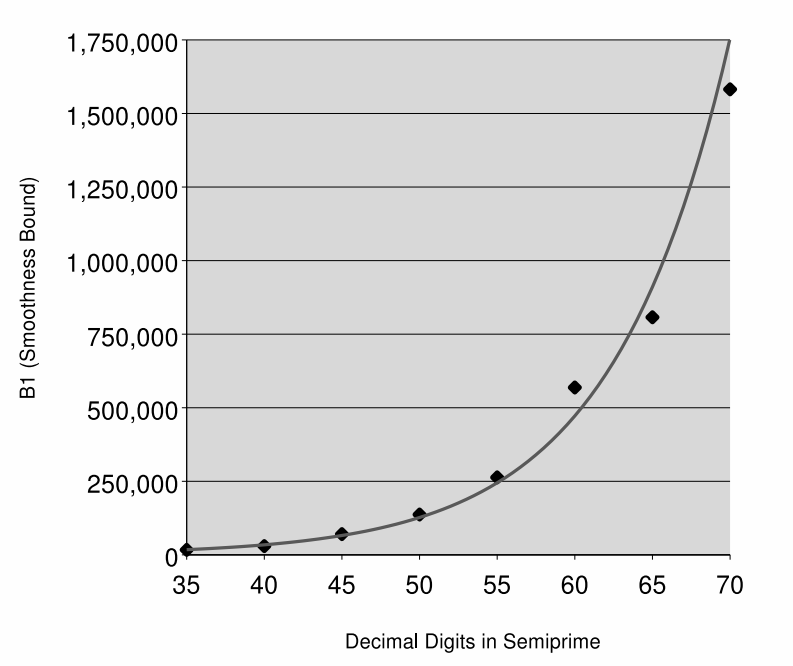

Figure 5.1 shows $T_{65}$ factored across a range of prime bounds and is characteristic of the test numbers. Despite many internal factors the runtimes behave in a consistent and parabola-like manner making for easy minimum location. This process is done for all numbers, moving in increments of 2.5% from Pomerance's $B_1$ suggestion until a minimum reached. Table 5.2 lists the optimum prime bounds for our implementation of basic QS arrived at via this method.

**Table 5.2**: Optimum $B_1$ Selection for Test Nums using Basic QS

| $N$ | Optimum-$B_1$ | Pomerance % | Density % |
|---|---|---|---|
| $T_{35}$ | 17,023 | 147.5% | 0.922% |
| $T_{40}$ | 29,723 | 115.0% | 0.553% |
| $T_{45}$ | 70,575 | 127.5% | 0.293% |
| $T_{50}$ | 137,134 | 120.0% | 0.164% |
| $T_{55}$ | 263,382 | 115.0% | 0.095% |
| $T_{60}$ | 568,835 | 127.5% | 0.046% |
| $T_{65}$ | 807,450 | 95.0% | 0.034% |
| $T_{70}$ | 1,582,110 | 100.0% | 0.020% |

Ironically, Pomerance's heuristic suggestion ends up being a decent one in the long run. Though the theory behind it does not hold in implementation, it is accurate because of a complex combination of factors difficult to quantify. By plotting the above points and applying an expotential least squares regression, we produce the graph of Figure 5.2

**Figure 5.2**: Optimum $B_1$ for Basic QS w/Expotential Regression




where the best fit function is given by

$$OptimumB(\ Digits\ ) = 178.952 + e^{0.131 * Digits} \tag{5.3}$$

and $Digits$ is the number of digits in the base 10 representation of $N$. Armed with this function, we compute the estimated $B_1$-values for the two larger integers $T_{75}$ and $T_{80}$ and run them with the calculated value, as well as $\pm 10\%$ of the same. If our central estimation produces a runtime inbetween or near the other two, we will consider our estimation sufficient. More rigorous testing would be ideal, but is prohibitively time consuming with integers this large (and thus why we wish to extrapolate in the first place). These tests produce the results shown in Table 5.3. One of the two test numbers lied within the desired

range, $T_{80}$, and evidence suggests the estimation for $T_{75}$ was not a horrible one. The range between our estimation for $T_{75}$ and the more accurate +10% of that was only 0.7% of the total runtime. This figure would seem to indicate our estimation is a good one; the small deviation tells us we must be near the minimum where the runtime parabola is rather flat. Meanwhile $T_{80}$ produced a very good result, beating the ±10% figures by over ten minutes apiece.

**Table 5.3**: Extrapolation Runs for Basic QS

| $N$ | Extrapolated-$B$ | $B$-runtime | $B - 10\%$ time | $B + 10\%$ time |
|---|---|---|---|---|
| $T_{75}$ | $3,384,660$ | $8,215.3$ | $8,195.9$ | $8,132.1$ |
| $T_{80}$ | $6,525,799$ | $30,309.6$ | $30,999.3$ | $31,063.4$ |

## 5.2 Large Prime Variations

Armed with some idea of how $B_1$ growth affects relation acquisition runtime, $B_2$ and LPVs are brought into the analysis. The situation is problematic. There is a lack of precise mathematical theory which can be brought to bear on LPVs. The new independent variable complicates matters immensely because the problem of runtime minimalization now resides in three dimensions. Should $B_1$ and $B_2$ be optimized separately, or should a ratio be used to describe the ideal relationship between the two? Lastly, are the ideal $B_1$ values found previously still useful when LPVs are introduced?

To begin, let us review what we know mathematically about LPVs. At the algorithms outset, $B_2$ is chosen such that $B_1 < B_2 < B_1^2$. When a $f(x_i)$ undergoes the trial-division verification procedure the cofactor, $C$, is examined. If $C < B_2$ and $C \neq 1$, then $f(x_i)$ is stored in hopes that another $f(x_i)$ with the same $C$ can be found and a smooth relation produced through combination of the two. So then, what is the probability of being "almost" $B_1$-smooth and a partial relation? It is hard to say, but one can certainly conclude Prob($B_1$-Smooth) $<$ Prob(LPV) $<$ Prob($B_1^2$-smooth). Finding an LPV is just half the battle, it is pointless to store a partial relation if another will never be encountered with

the same cofactor. As Pomerance [3] notes, the birthday paradox gives good probability that partial relation pairs will be encountered. However, very large primes are unlikely to ever find a match and the text suggests that $B_2$ should lie on the interval $20B_1 \ldots 100B_1$.

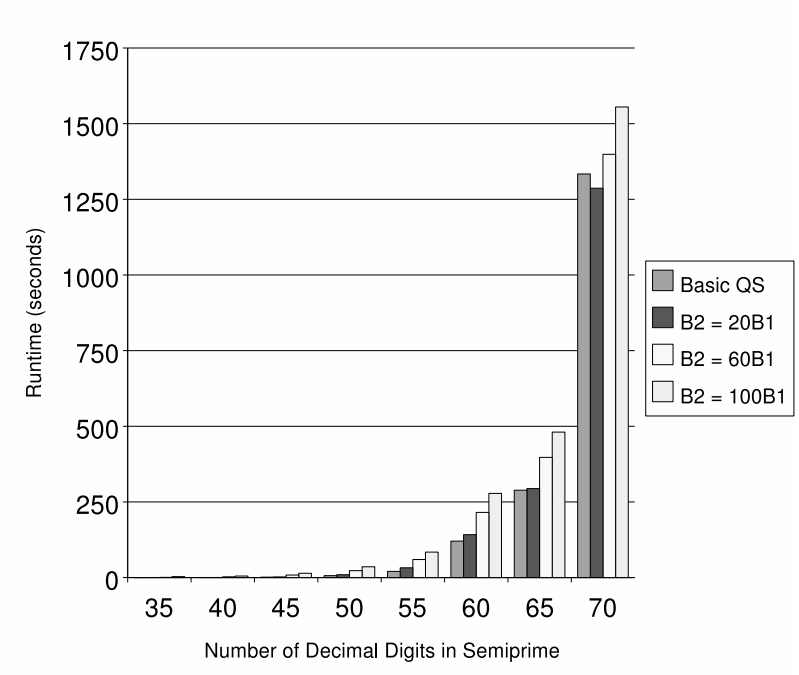
From the implementation perspective, examining the effects of $B_2$ sizing is much simpler than with $B_1$ above. The variable is relied upon infrequently for other purposes. A quick review of its role should suffice. First, recall that $B_2$ sizes the large prime bases. All primes $(B_1, B_2]$ are located and stored. They are run through the Legendre test, but residue solve values do not have to be located. Unlike smooth primes, where the primes are just stored, each large prime has 5 different fields. Not only is there the prime, but also a boolean, and three mpz_ts to store a partial relation. At the algorithm's end the memory requirements can be large. Not only is the exponent matrix large and populated, but many of the partial relation slots will be filled. In the early days of QS, some researchers like those at Sandia National Labs [5] abandoned LPVs altogether because the memory requirements were so strenuous. Even though the range of $B_1 \ldots B_2$ may be large, the logarithmic growth of the prime counting function alleviates some memory concerns.

With the installation of LPVs comes overhead regarding their location, storage, and communication. With a larger error bound in place (which $B_2$ sizes) it means more numbers undergo expensive trial division verification. Each full relation produced as a combination of partials requires two such divisions, and do not forget about partials which are stored but never encounter a matching cofactor. For these, the time spent in trial division is pointless. Then there are communication costs. In the parallel version, partial relations are communicated just like a full relation. All the trouble of converting and packing must be done more often since partials will now occur in addition to smooth relations.

For small numbers where smooth relations are easier to find, the LPV strategy might be fruitless. Another objective, then, is to determine where crossover occurs for LPVs being beneficial. Also consider that by ignoring the linear algebra portion of the algorithm, one loses one of the most beneficial aspect of LPV's; the ability to keep exponent vector length small while still finding relations quickly.

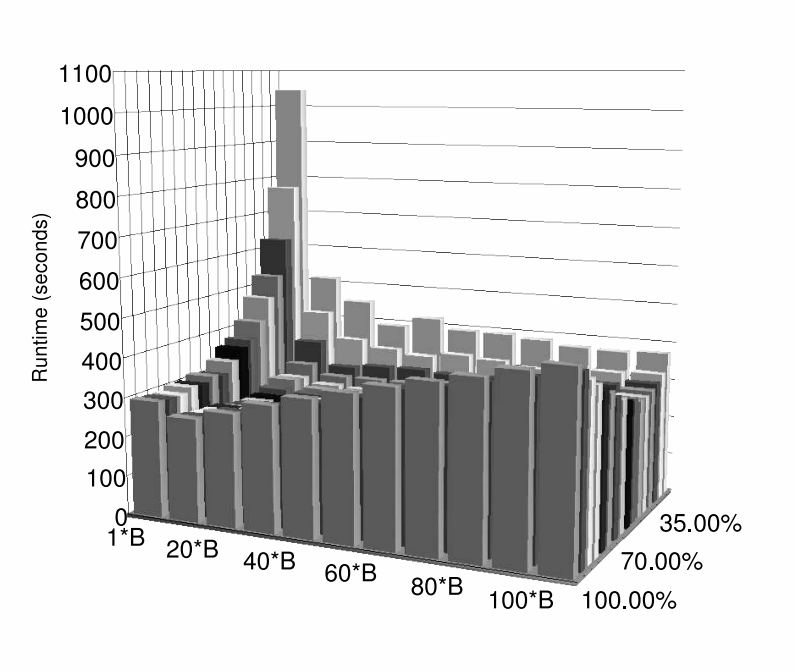
With little theory to guide us, it makes sense to dive straight into runtime analysis. From this point forward, we will use the term *bound ratio* to describe the relationship between $B_1$ and $B_2$ such that $B_1 * \text{bound-ratio} = B_2$. This allows for a more concise description of the relationship between the two numbers. Using the optimum $B_1$ values we calculated above for basic QS, we will test the extremities and median of Pomerance's suggestion using ratios of 20, 60, 100 and examine the results.

**Figure 5.3**: Pomerance's LPV Suggestions v. Basic Runtimes




As Figure 5.3 shows, with these parameters, only the largest test number produces any runtime improvement whatsoever. This seems contradictory to what we know about large prime variations. Perhaps we should try reducing the size of $B_1$ and thereby reducing the number of relations to be found, as well as communications and memory requirements. Figure 5.4 presents $T_{65}$ factored across a variety of values in both $B_1$ and $B_2$ dimensions. Just as with basic QS, the graph is well behaved for minimum location (considering we are now working in 3 dimensions). Using iterative methods and a large number of test runs the

**Figure 5.4**: $T_{65}$ Acquisition Runtimes w/Varying $B_1$ and $B_2$




optimum $B_1$ and *bound ratio* which produce the absolute minimum runtime are calculated for each test number. Reductions from basic QS $B_1$ are done in 2.5% increments, and *bound-ratio* is altered in increments of 5. With this in place, our findings are illustrated in Table 5.4.

**Table 5.4**: Optimum $B_1$, $B_2$ Selection for Test Nums using QS w/LPV

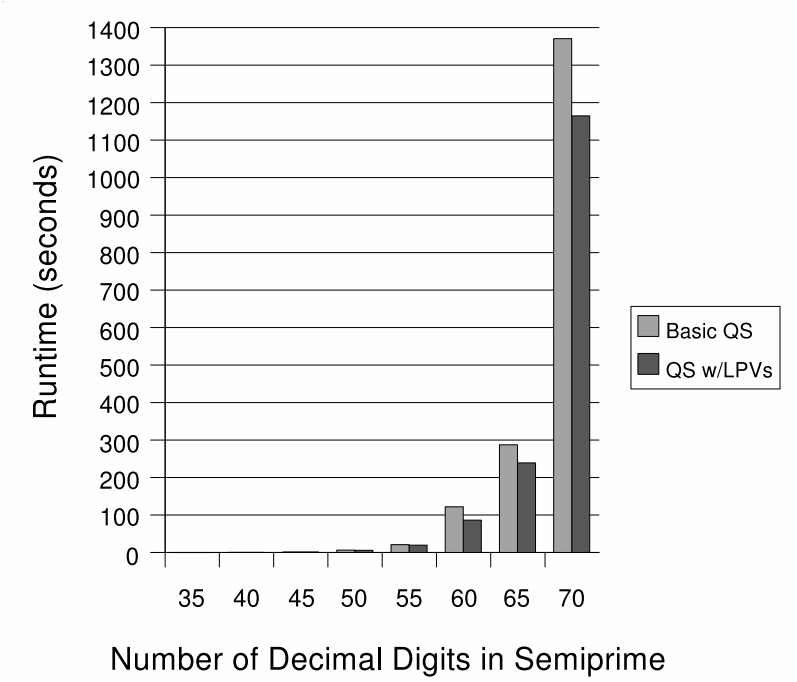
| $N$ | $B_1$ | % of Basic $B_1$ | B-Ratio | Density |
|---|---|---|---|---|
| $T_{35}$ | 17,023 | 100% | 1 | 0.922% |
| $T_{40}$ | 29,723 | 100% | 1 | 0.553% |
| $T_{45}$ | 49,403 | 70% | 5 | 0.503% |
| $T_{50}$ | 82,280 | 60% | 20 | 0.341% |
| $T_{55}$ | 184,267 | 70% | 10 | 0.161% |
| $T_{60}$ | 312,859 | 55% | 10 | 0.108% |
| $T_{65}$ | 524,843 | 65% | 10 | 0.064% |
| $T_{70}$ | 1,107,477 | 70% | 40 | 0.033% |

The reduction in $B_1$ values from basic QS seems to happen at a pretty consistent rate of between 55% and 70% for significantly sized integers. Unfortunately, $B_2$ demonstrates

erratic behavior making any type of regression pointless. If more resources were available (e.g. time) it would be ideal to factor several semiprimes of approximately the same length to see if these figures were consistent across similarly sized numbers. This could give some idea of how variable runtime truly is as a result of the algorithm's Las Vegas tendencies.

Factorizations of $T_{75}$ and $T_{80}$ were attempted using the LPV strategy but failed due to a lack of memory. This raises another interesting point. How can memory concerns be dealt with when factoring huge integers? Factoring $T_{80}$ with $B_1$ equal to 65% of the extrapolated value from basic QS results in $B_1 = 6,525,799$ and with BRatio= 20 then $B_2 = 130,515,980$. Even with the logarithmic growth of the prime counting function this an absolutely huge number of large primes (and therefore, potential partial relations) that have to be stored. The only viable solution seems to be the use of disk storage. Structures like the smoothArray, residueArray, logArray, partArray, and hasPartial should be kept in physical memory, but the others should be written to file and accessed only as needed.

**Figure 5.5**: Comparing Optimum Acquisition Runtimes w/Basic QS and PQS




Despite the huge memory overhead of using LPVs, our findings did show the strategy

offered time savings over basic QS, as Figure 5.5 shows. The resulting matrix was smaller in dimension, as well, though it was approximately twice as dense.

Our conclusions seem to support Pomerance's suggestion that bound selection is as much of an art as a science. While a general range of estimation is possible, true optimization seems unlikely in an algorithm with variable runtimes and many internal factors dependent on user-selected variables.

# Chapter 6

# Conclusion

Recall the two objectives from the introduction: First, to construct and describe an efficient Quadratic Sieve (QS) implementation, and secondly to utilize this implementation in bound determination. While optimization is not quantifiable, our solution has undergone many refinements and uses well established speedup techniques (constant time block population, logarithmic estimation). That being said, our implementation does not intend on setting any factorization records. Notable improvements (multiple polynomials, self-initialization) are ignored to promote simplicity and focus on the research question with minimal independent variables. The well-documented source code would, however, act as an excellent foundation for those wishing to experiment with QS.

Concerning bound selection; both theory and runtime analysis were combined to produce a function that optimizes bound determination for the sieving phase of basic QS. The validity of this production is supported by the large test data set and accuracy of small extrapolations. The same cannot be said for bounds when using LPVs. With the problem lying in three dimensions, optimized bounds could be located for individual test numbers. However, there seemed to be no pattern as to how bounds developed as numbers grew larger, making extrapolations impossible. Additional testing is necessary to determine if this randomness is inherent in the algorithm or the result of some property of our randomly chosen test numbers. Further refining the implementation (multiple polynomials, for ex-

ample) may alter ideal bounds, but the ideas provided herein provide some basis for their determination.

## 6.1 Significance of Factorizations

Even in the advanced state of the art, there exists no factorization algorithm with polynomial runtime complexity for classical computing models. This fact forms the basis of several modern cryptography systems, most notably the RSA public-key encryption algorithm. Because of the super-polynomial complexities for currently known methods even marginal increases in key-length result in large security gains. This has permitted the algorithm to remain relevant without the need for absurdly large keys, even with hardware improvements.

Because of the aforementioned growth factors, current decomposition methods will never be practical as $N$ grows very large. Hardware improvements have contributed as much, if not more, to our capability to factor "large" numbers than theoretical advances. If a (classical) polynomial time algorithm does exist, it probably relies heavily on existing techniques. Thus it seems appropriate to devote research to the study of modern "inpractical" methods. A polynomial time algorithm does exist for quantum computers, known as Shor's algorithm. The non-existence of stable and large-scale quantum computer, however, keeps this work largely in the theoretical realm.

## 6.2 Future Work/Shortcomings

Aside from the obvious improvements to the algorithm (double large prime variations, multiple polynomials), the work of this paper introduces some more subtle research topics. Foremost, runtimes of an efficient parallel matrix elimination implementation are needed with regards to dimension and density. This has been done before but trying to scale timings based on hardware differences is a difficult task. Only if sieving and elimination statistics come from the same machine can optimization truly be achieved. While some

authors concern themselves only with asymptotic bounds and heuristic estimations, these simply do not seem appropriate for problems of this magnitude.

Secondly, our program and discussion have abstracted the workings of the GMP library functions. While in design they are general purpose mathematical functions, our algorithm often utilizes them for specific purposes. The average method is full of code checking arguments and conditionals handling special cases that may not always be necessary. Consider the case of mpz_remove(·) which is called many times during probable relation verification. Could the fact we always attempt to divide by primes somehow improve the function? Since GMP is an open-source library it is very easy to tinker with such improvements. For that matter, there is no evidence that GMP is even the best library for these purposes. While it claims to be the fastest bignum library there are several number theory specific libraries which could improve upon the subset of GMP functions most critical to QS.

In relation to the topic of multiple-precision integers is the scalability of the current implementation. It is ideal to use the smallest possible datatypes wherever possible to speed the algorithm. Yet, some type choice decisions are ambiguous. For example, the unsigned long integer limitation on primes (both smooth and large) could be insufficient when factoring across a very large parallel machine or with much faster hardware. While storing and basic manipulation of primitive types is advantageous some GMP methods accept only mpz_t integers. Although type casting is sometimes required, this could be remedied if new GMP functions are authored.

It is known the two major bottlenecks of the algorithm are the sieving and matrix elimination stages. Consider a strategy where the runtimes for each can be reduced in the general case, which we will call *continuous matrix elimination*. Only in the worst case is it necessary to find greater than numSmoothP relations in order to produce a zero vector (congruence of squares). Probablistically speaking, many fewer relations are needed. It is possible, albeit unlikely, for a single relation to have a zero vector and thus be a perfect square. Why not intermediately perform matrix elimination during the sieving stage in the hopes a dependecy already exists? Elimination will be quick(er) since not all vectors

have been found and runtime correlates to the probability of finding a solution. This applies exceptionally well to a parallel environment. A single node, or group of nodes, could dedicate themselves to matrix elimination while the remainder continue sieving. If an elimination produces a non-trivial congruence (solution), the whole algorithm can quit immediately. Otherwise, the elimination node(s) can receive the exponent vectors found during the last elimination, add them to the matrix, and eliminate again; proceeding until a solution is found. It would be ideal if the master node were a multiprocessor machine with shared memory, this way elimination could be performed without having to communicate the potentially large matrix and blocking other communications. In theory, use of this probablistic runtime strategy could produce quick terminations for very large numbers.

Finally, some additional research ideas are presented in brief: (1) Ideal sizing of variable BLOCKSIZE is another interesting question since most authors choose to address the topic without regard for estimated block population strategies. (2) Assembly language could be used to optimize the algorithm's most frequently used inner-loops. (3) One could take the OEIS sequence of section (3.3.3), compensate for constant $N$, and develop a block population strategy far more accurate for small $f(x_i)$. Lastly, (4) the effectiveness of re-sieving a block using higher powers of primes could be evaluated.

# Appendix A

# Test Numbers and Protocol

The following are the test semiprimes referenced throughout the paper. They are of the form $T_D$ where $D$ is the number of digits in the decimal representation. Numbers are chosen such that they are within $\pm 2\%$ of the digit median. The factors are randomly chosen probable primes passing 10 Miller-Rabin tests, which when multiplied, meet the above criteria.

**Table A.1**: Test Semiprimes

| | |
|---|---|
| $T_{35}$ | $49571111159984591898201756935135699 =$<br>$390587352976158827 * 126914276108193337$ |
| $T_{40}$ | $5134627416122322171946874768462846705393 =$<br>$59381292121142992993 * 86468772111715526801$ |
| $T_{45}$ | $5154946366005870274049600766421991992307163 23 =$<br>$11133935136461447082881 * 46299410790749400981283$ |
| $T_{50}$ | $5109458780159919103652505460182146005123600461 4739 =$<br>$65282204365047241650385 63 * 78267252612805082772543 53$ |
| $T_{55}$ | $5161709491005113940049767551967754724363596958307983011 =$<br>$52801965409982808772353814 81 * 97756010612991962222528913 1$ |
| $T_{60}$ | $5085693047229728635447320757814311009260913163208871966793 63 =$<br>$730242536385745613521330054849 * 69643889445290354179516046758 7$ |
| $T_{65}$ | $5091605983718010722793535920042252613457285711265552759327967570 1 =$<br>$12403352833232528532453062001370 1 * 41050239013405942383766007026200 1$ |
| $T_{70}$ | $5022455732342638103259825578365273221474503776819722194915601821957969 =$<br>$59460903673872461055142471844937653 * 84466522067836322644451953206848173$ |

It would be advantageous if test numbers could be considered in 1-digit increments. However, such refinement becomes inpractical given the amount of testing performed on each

number. The larger semiprimes below are not in the function-generating data set but used to determine the accuracy of the functions for small extrapolations.

**Table A.2**: Large Test Semiprimes (For Extrapolations)

| | |
|---|---|
| $T_{75}$ | 519908592343823838897189709663477114588903953712914495843508668302129843291 = 225984105285464814278921524350915964 31 * 230064230263925602755133650051101310 61 |
| $T_{80}$ | 505298868893099620951298292716901978142916525885495874940323093953346750701117 33 = 778465374909555969322334751853358268313 1 * 649096138607072269623077829406411690414 3 |

Because of background communications and system process variations among nodes there is some runtime deviation when performing identical tests at different times. For this reason, critical test cases are run three times and the average of each is that plotted. For all runs BLOCKSIZE= $100,000$. Though not the most advantageous in all cases it is a median selection for a variable that needs held constant. If not explicitly mentioned otherwise, parallel runs and timings were performed on 32 nodes.

All work was done in C/C++ and compiled with the -O3 optimization flag. All testing and timings are performed on The Inferno, Washington and Lee University's 64-node Beowulf cluster. Each node contains a 3.0 GHz Pentium IV processor with 1GB of physical memory. The nodes have switch interconnectivty across a Gigabit Ethernet. Funding for the cluster includes internal grants from Washington and Lee University and external grants from the Jeffress and Keck foundations.

# Appendix B

# Psuedocode

```
========================== SEQUENTIAL VERSION ================================

ulong smoothArray[]   // Contains smooth primes w/Legendre == 1
ulong residueArray[]  // Contains lower quadratic residue for above primes
float logArray[]      // Contains log() of all smoothArray primes

ulong partArray[]     // Contains large primes w/Legendre == 1
bool  hasPartial[]    // Indicates if large prime has stored partial rel.
mpz_t partX[]         // Contains x portion of partial rel.
mpz_t partFX[]        // Contains f(x) portion of partial rel.
mpz_t partROW[]       // Contains exponent vector portion of partial rel.

mpz_t relationX[]     // Stores x portion of smooth rel.
mpz_t relationFX[]    // Stores f(x) portion of smooth rel.
mpz_t expMatrix[]     // Matrix formed from exponent vectors of all smooth rels.

float curBlock[]      // Block currently being sieved/examined


quadSieveLPVSeq( mpz_t N, ulong smoothBound, float BRatio ) :
    ulong numSmoothP  // Number of (smooth) primes in smoothArray
    createSmoothArrays( smoothBound )
    numSmoothP = fillArrays( smoothBound, N )
    createLogArray( numSmoothP )
    ulong numPartP // Number of (large) primes in partArray
    ulong partBound = smoothBound * BRatio
    numPartP = createPartArray( N, smoothBound, partBound )
    createPartStructs( numPartP )
    createRelStructs( numSmoothP )
    findRelations( N, partBound, numSmoothP )
    augmentIndentity( expMatrix )
    matrixEliminate( expMatrix )
```

```
    processZeroRows( expMatrix )

createSmoothArrays( ulong smoothBound ) :
    smoothEst = overestimatePrimes( 0, smoothBound )
    smoothArray = create array [ smoothEst ]
    residueArray = create array [ smoothEst ]

fillArrays( ulong smoothBound, mpz_t N ):
    smoothArray[0] = 2
    residueArray[0] = 1
    ulong numSmoothP = 1
    ulong curPrime = 3
    while curPrime <= smoothBound :
      if legendre(curPrime, N) == 1 :
        smoothArray[numSmoothP] = curPrime
        residueArray[numSmoothP] = findResidue( curPrime, N )
        numSmoothP++
      curPrime = nextPrime( curPrime )
    return numSmoothP

createLogArray( ulong numSmoothP ) :
    logArray = create array [numSmooth]
    for i = 0 to numSmooth - 1 :
      logArray[i] = log(smoothArray[i])

createPartArray( mpz_t N, ulong smoothBound, ulong partBound) :
    ulong partEst = overestimatePrimes( smoothBound+1, partBound )
    partArray = create array [partEst]
    while curPrime <= partBound :
      if legendre(curPrime, N) == 1 :
        partArray[numPartP] = curPrime
        numPartP++
      curPrime = nextPrime(curPrime)
    return numPartP

createPartStructs( ulong numPartP ) :
    hasPartial = create array [numPartP]
    partX = create array [numPartP]
    partFX = create array [numPartP]
    partROW = create array [numPartP]

createRelStructs( ulong numSmoothP ) :
    ulong relCapacity = 10 + numSmoothP
    relationX = create array [relCapacity]
    relationFX = create array [relCapacity]
    expMatrix = create array [relCapacity]

findRelations( mpz_t N, ulong partBound, ulong numSmoothP ) :
    mpz_t blockStart = sqrt(N)
    mpz_t upperBound = sqrt(2N)
    ulong relsFound = 0
```

```
    curBlock = create array [BLOCKSIZE]
    while blockStart <= upperBound && relsFound != (10+numSmoothP) :
      populateBlock( blockStart, BLOCKSIZE )
      sieveBlock( blockStart, BLOCKSIZE )
      testProbs( &relsFound, partBound, numSmoothP )
      blockStart += BLOCKSIZE

populateBlock( mpz_t blockStart, ulong BLOCKSIZE ) :
    for i = 0 to BLOCKSIZE - 1 :
      curBlock[i] = numBitsIn(f(blockStart+(BLOCKSIZE/2)), N)

sieveBlock( mpz_t blockStart, ulong BLOCKSIZE ) :
    for i = blkEntry( blockStart, 2, 1 )  to BLOCKSIZE - 1 by 2 :
      curBlock[i] -= logArray[0]
    for i = 1 to numSmoothP - 1 :
      ulong prime = smoothArray[i]
      ulong residue = residueArray[i]
      for j  = blkEntry( blockStart, prime, residue ) to BLOCKSIZE - 1 by prime:
        curBlock[j] -= logArray[i]
      residue = prime - residue
      for j  = blkEntry( blockStart, prime, residue ) to BLOCKSIZE - 1 by prime:
        curBlock[j] -= logArray[i]

testProbs( ulong relsFound, ulong partBound, ulong numSmoothP ) :
    mpz_t expRow
    float errorBound = log(partBound)
    for i = 0 to BLOCKSIZE - 1 :
      if curBlock[i] < errorBound :
        ulong cofactor = relVerify( f(blockStart+i), numSmoothP, expRow)
        if cofactor == 1 :
          relationX[relsFound] = blockStart + i
          relationFX[relsFound] = f(blockStart + i)
          expMatrix[relsFound] = expRow
          relsFound++
        else if cofactor < partBound :
          ulong index = findCofactor( cofactor ) // Binary search in partArray
          if hasPartial[index] :
              // See section 2.5 on combining partials to make smooth rels.
            relationX[relsFound] = makeSmoothX(blockStart+i, partX[index])
            relationFX[relsFound] = makeSmoothFX(f(blockStart+i), partFX[index])
            expMatrix[relsFound] = expRow XOR partROW[index]
            relsFound++
          else :
            partX[index] = blockStart + i
            partFX[index] = f(blockStart + i)
            parROW[index] = expRow
            hasPartial[index] = true

============================ HELPER METHODS ==================================

f( mpz_t x, mpz_t N ) :
```

```
    return x^2-N

findResidue( mpz_t N, ulong prime ) :
    for i = 1 to prime / 2 :
      if i^2 % prime == N % prime :
        return i

relVerify( mpz_t probRel, ulong numSmoothP, mpz_t &expRow ) :
    for i = 0 to numSmoothP - 1 :
      if(( mpz_remove(probRel, probRel, smoothArray[i]) % 2 ) == 1 ) :
        mpz_setbit( expRow, i )
    return probRel

blkEntry( mpz_t blockStart, ulong prime, ulong residue ) :
    ulong startPos = -1 * (blockStart/prime) + residue + prime
    if startPos > prime :
      starPos -= prime
    return startPos

============================ PARALLEL VERSION ================================


ulong smoothArray[]   // Contains smooth primes w/Legendre == 1
ulong residueArray[]  // Contains lower quadratic residue for above primes
float logArray[]      // Contains log() of all smoothArray primes

ulong partArray[]     // Contains large primes w/Legendre == 1
bool  hasPartial[]    // Indicates if large prime has stored partial rel.
mpz_t partX[]         // Contains x portion of partial rel.
mpz_t partFX[]        // Contains f(x) portion of partial rel.
mpz_t partROW[]       // Contains exponent vector portion of partial rel.

mpz_t relationX[]     // Stores x portion of smooth rel.
mpz_t relationFX[]    // Stores f(x) portion of smooth rel.
mpz_t expMatrix[]     // Matrix formed from exponent vectors of all smooth rels.

float curBlock[]      // Block currently being sieved/examined


quadSieveLPVPar( mpz_t N, ulong smoothBound, float BRatio ) :
    if nodeNum == 0 :
      rootNode( N, smoothBound, BRatio )
    else :
      computeNode()

rootNode( mpz_t N, ulong smoothBound, float BRatio ) :
    broadcast( N )
    ulong partBound = smoothBound * BRatio
    broadcast( partBound )
    createSmoothArrays( smoothBound )
    numSmoothP = fillArraysRoot( smoothBound, N )
```

```
    createLogArray( numSmoothP )
    bcastStructs( numSmooth P )
    ulong partBound = smoothBound * BRatio
    ulong numPartP = createPartArray( N, smoothBound, partBound )
    createPartStructs( numPartP )
    createRelStructs( numSmoothP )
    findRelationsRoot( N, partBound, numSmoothP )
    augmentIndentity( expMatrix )
    matrixEliminate( expMatrix )
    processZeroRows( expMatrix )
    cleanupMessages()

computeNode() :
    broadcast( mpz_t N )
    broadcast( ulong partBound )
    fillArraysCompute( N )
    broadcast( ulong numSmoothP )
    recvSmoothArrays( numSmoothP )
    findRelationsCompute( N, partBound, numSmoothP )

fillArraysRoot( ulong smoothBound, mpz_t N ) :
    smoothArray[0] = 2
    residueArray[0] = 1
    ulong numSmoothP = 1
    ulong curPrime = 2
    int quitNodes = 0
    while quitNodes != numProcs-1 :
      curPrime = nextPrime( curPrime )
      while legendre(curPrime, N) != 1 :
        curPrime = nextPrime( curPrime )
      receive( ulong prime, ulong residue )
      if prime != 0 :
        smoothArray[numSmoothP] = prime
        residueArray[numSmoothP] = residue
        numSmoothP++
      if curPrime <= smoothBound :
        send( curPrime )  // Send to node who just returned residue
      else :
        send( quit )
        quitNodes++
    return numSmoothP

fillArraysCompute( mpz_t N ) :
    send( 0 ) // Request data from root node
    while message.tag != quit:
      receive( ulong prime )
      send( findResidue( prime, N ) )

bcastStructs( ulong numSmoothP ) :
    broadcast( numSmoothP )
    broadcast( smoothArray )
```

```
    broadcast( residueArray )
    broadcast( logArray )

recvSmoothArrays( ulong numSmoothP ) :
    smoothArray = create array [ numSmoothP ]
    residueArray = create array [ numSmoothP ]
    logArray = create array [ numSmoothP ]
    broadcast( smoothArray )
    broadcast( residueArray )
    broadcast( logArray )

findRelationsRoot( mpz_t N, ulong partBound, ulong numSmoothP ) :
    ulong relsFound = 0
    int quitNodes = 0
    while relsFound != numSmoothP+10 && quitNodes != numProcs-1 :
      receive( message )
      if message.tag == quit :
        quitNodes++
      if message.tag == smooth :
        unpack( mpz_t x, mpz_t row from message )
        relationX[relsFound] = x
        relationFX[relsFound] = f( x )
        expMatrix[relsFound] = row
        relsFound++
      else :
        unpack( ulong cofactor, mpz_t x, mpz_t row from message )
        ulong index = findCofactor ( cofactor )
        if hasPartial[index] :
            // See section 2.5 on combining partials to make smooth rels.
          relationX[relsFound] = makeSmoothX( x, partX[index] )
          relationFX[relsFound] = makeSmoothFX( f(x), partFX[index] )
          expMatrix[relsFound] = row XOR partROW[index]
          relsFound++
        else :
          partX[index] = x
          partFX[index] = f( x )
          parROW[index] = row
          hasPartial[index] = true

findRelationsCompute( mpz_t N, ulong partBound, ulong numSmoothP ) :
    mpz_t blockStart = sqrt(N) + BLOCKSIZE * nodeNum
    mpz_t upperBound = sqrt(2N)
    curBlock = create array [BLOCKSIZE]
    while blockStart <= upperBound :
      populateBlock( blockStart, BLOCKSIZE )
      sieveBlock( blockStart, BLOCKSIZE )
      testProbsCompute( partBound, numSmoothP )
      iprobe( bool isMessage)
      if isMessage :
        quit
      blockStart += BLOCKSIZE * numProcs
```

```
    send( quit )

testProbsCompute( ulong partBound, ulong numSmoothP ) :
    mpz_t expRow
    float errorBound = log(partBound)
    for i = 0 to BLOCKSIZE - 1 :
      if curBlock[i] < errorBound :
        ulong cofactor = relVerify( f(blockStart+i), numSmoothP, expRow)
        if cofactor == 1 :
          pack( blockStart+i, expRow into buffer)
          send( buffer )
        else if cofactor < partBound :
          pack( cofactor, blockStart+i, expRow into buffer )
          send( buffer )

cleanupMessages() :
    while quitNodes != numProcs-1 :
      receive( message )
      if message.tag != quit :
        send( quit ) // Send to node who we just received from
      else :
        quitNodes++
```

# Bibliography


[1] Olof Åsbrink and Joel Brynielsson. Factoring large integers using parallel quadratic sieve. Technical report, Royal Institute of Technology, Sweden, April 2000.

[2] CACAO and Arenaire project teams. MPFR library. online: http://www.mpfr.org, 2006.

[3] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*. Springer-Verlag, New York, 1st edition, 2001.

[4] Free Software Foundation. GMP: Arithmetic without limitations. online: http://www.gmplib.org, 2007. Source and documentation for the GNU multiple precision arithmetic library.

[5] Diane B. Holdridge and James A. Davis. Factorization using the quadratic sieve algorithm. Technical Report SAND83-1346, Sandia National Laboratories, Albuquerque, NM, December 1983.

[6] Gina Kolata. 100 quadrillion calculations later, eureka!; a math problem is solved years ahead of expectations. *New York Times*, page A13, April 27 1994.

[7] RSA Laboratories. RSA-200 is Factored! online: http://www.rsa.com/rsalabs/node.asp?id=2879, May 2005.

[8] Carl Pomerance. The quadratic sieve factoring algorithm. In *Advances in Cryptology, Proceedings of EUROCRYPT '84*, T. Beth, N. Cot, and I. Ingemarsson, editors, pages 169–182, Berlin, 1984. LNCS 209, Springer-Verlag.

[9] N. J. A. Sloane. The on-line encyclopedia of integer sequences. online: www.research.att.com/ njas/sequences/, 2007.

[10] Eric W. Weisstein. MathWorld - a Wolfram web resource. online: http://www.mathworld.com.