Thesis

Ammon Bartram

Introduction

All positive integers have a unique decomposition into primes. This Fundamental Theorem of Arithmetic has been known since the time of the ancient Greeks. Its proof, however, is purely existential; it tells us nothing about how to efficiently find these unique primes. For most of the last two thousand years no one cared. The Fundamental Theorem of Arithmetic was a theoretical issue, and factoring poses no theoretical problem. If we divide an integer _n_ by every number from _2_ through _n/2_ in turn and check for a remainder, we will find all factors of _n_. In fact, we can easily improve on this by noting that if _n_ is composite it must have a factor less then _ceil(sqrt(n))_. In practice, however, even this second form of trial division is hopelessly inefficient. The number of divisions required is exponential in the length of the encoding of _n_. And as _n_ grows, this solution quickly becomes intractable.

The difficulty of the factoring problem became more than a theoretical issue in the twentieth century. The new fields of computational and coding theory brought attention to the study of algorithms, and made prime factorizations of practical importance. Specifically, it was realized that a practically calculable (polynomial time) homomorphism between multiset of primes numbers and their products would be a powerful tool (we see the power of such encoding in Gödel numbering). And factoring was strongly linked to the discrete logarithm problem and shown equivalent to the calculating modular square roots. The largest boost to interest in factoring, however, came with the development of public-key cryptography in the 1070s. RSA, one of the first public-key cryptographic systems discovered and still a workhorse of computer security, is based in the presumed difficulty of factoring large integers. If we can factor integers we can break RSA. This fact led to the creation of a factoring challenge with significant monetary prizes, and prompted a flurry of work around factoring.

Throughout the 60s and 70s, researchers working with ides of Fermat and others and armed with ever more powerful computers, managed to push the size of factorable number from 25 digits into the 40s. While vast improvements over trial division, these algorithms still suffered from the exponential relationship between the length of the number and the number of steps needed to find a factor. Before larger numbers could be factored, something fundamental had to change.

The Quadratic Sieve:

The Quadratic Sieve was the first algorithm to clearly break this exponential relationship. Developed by American mathematician Carl Pomerance in 1981, it built on previous factoring methods, combining abstract and linear algebra with the a time-space tradeoff to factor general integers (integers with no special form) in a subexponential number of steps. And its time-dominant step, location smooth numbers in a polynomial progression, is easily carried out in parallel across multiple machines. This algorithm quickly set new records, more than doubling the length of number that could be reliably factored in a span of 5 years. The QS is no longer the fastest general factoring algorithm. Since 1981, two other subexponential algorithms have been developed. Neither of these algorithms entirely replaces the QS, however. The Elliptical Curve Method, developed in 1987 by Hendrik Lenstra, has the same time complexity as the QS, but as

a function of the smallest factor of n instead of n itself. Thus it is good at finding small factors even of enormous numbers. But it requires higher precision calculations, and thus is less efficient than the QS for numbers with large factors. The General Number Field Sieve, in contrast, surpasses the QS only on very large numbers. Its asymptotic time complexity is less but it has a larger constant time, and thus is only more efficient for number with more than 110 digits. For semiprime integers (the product of two large primes) less than 110 decimal digits, the Quadratic Sieve remains the best known algorithm.

Our Research:

Carl Pomerance is a mathematician, and he developed the QS at a time when its memory use required a multi-million dollar supercomputer. Consequently, when writing his landmark papers in the early 1980s, he had never implemented the algorithm. His work is theoretical in nature, and he particularly gives very little attention to parallelizing the algorithm. The aim of our research is to back up Pomerence's work with empirical data. We are also interested in exploring the process of taking theoretical work and distilling it down to functioning code. We want to provide data documenting the runtime complexity of the algorithm, testing Pomerence's assertions about optimal parameter selection, and investigating the effects and efficiency of parallelizing the algorithm.

With these goals, we have spent the last year developing a parallel implementation of the QS in Java. This document presents the results of that research. To provide context, we first give a detailed description of the basic algorithm. Then, we present Pomerence's analysis of the runtime complexity of the QS along with the related problem of optimal parameter selection. Next we explain several algorithmic optimizations that we have implemented. And finally, we present the data that we have gathered with our implementation and the conclusions that this data supports.

Sub exponential Factoring and the Quadratic Sieve

The Quadratic Sieve revolutionized factoring, but it did not come from a vacuum. Rather, Pomerence developed it as an improvement to preexisting methods. In order to understand the Quadratic Sieve, then, we need to understand these older methods.

Trial Division:

The obvious way to factor an integer N is to divide it by all numbers x, 2 <= x <= n/2 (of course we only have to divide by primes, but as finding primes is superlinear this is not a saving). We can improve on this by noting that if x is a factor of N, then either x or its cofactor (N/x) must be less than or equal to ceil(sqrt(N)), so we really only have to search this smaller space. For small numbers, this does not pose a problem. And for large numbers, we can at least search part of the space. Thus trial division is sufficient for factoring small numbers or finding small factors of large numbers. For general large integers, however, it is intractable. Trial division of a 56 digit numbers at 10^6 divisions per second would take around 13 billion years to finish, or the entire history of our universe.

Fermat's Method:

Fermat tried to improve on trial division by representing N as a difference of squares. Then N = (u^2 – v^2) = (u+v)(u-v) gives a factorization. This is not a special case. All composite integers can be expressed as a difference of squares. Unfortunately, finding such squares is no

more efficient in the general case than trial division. It splits integers with factors close to their square root more efficiently, but its worse case if worse than the worse case of trial division.

We note here that Fermat's Method only splits N, and does not find a prime factorization. However, the relative ease of finding small factors is such that semiprimes (the product of two primes) are the hardest case of the factoring problem. Thus splitting in fact suffices. This can be seen by noting that after we have extracted the smallest prime factor p from N and are left with cofactor C, we know that C has no factors less than p. Thus we do not have to redo any division, and the bound on our search space is now sqrt(C) which is less than the original sqrt(N). This is only a proof in the case of factoring by trial division, but the result holds for all known algorithms, and from now on we will consider finding a single factor equivalent to solving the factoring problem.

Kraitchik's Method

In 1925, Belgian mathematician Maurice Kraitchik realized that it is sometimes sufficient to take a more general case of the equality from Fermat's Method. Specifically, he realized that it is sometimes enough to find _(u^2 – v^ 2) = kN for some integer k, or u^2 = v^2 (mod N). This gives N | (u-v)(u+v). Now, if it also happens to be the case that u != +-v (mod N) then by the same reasoning N!|(u+v) and N!|(u-v). Thus we have N divides the product _xy_ but does not divide x and does not divide y. This is only possible of N is composite, and shares non-trivial (not 1 or N) factor with x and a non-trivial factor with y. We conclude that gcd(N, u-v) is a non-trivial factor of N.

Now, we must examine the above condition. The factorization only follows from the condition u != +-v (mod N). We will now show that this is not rare. Indeed, as long as N has two unique odd prime factors, then at least half of all existing pairs of congruent squares u^2 = v^2 (mod N) will give u != +-v (mod N).

Proof:
Let u^2 = v^2 (mod N) with u = +- v. Assume without loss of generality that -N/2 < u, v <= N/2 (u and v are representative elements of their congruence classes, with elements over N/2 reduced by N). Then, if u =+- v (mod N), either u = v or u = – v. That is, u = vk, where k is 1 or -1. All pairs of congruent squares that yield trivial factorizations, then, belong to the set s = {(x^2, y^2): x in [u] and y in [v] with -N/2 < u, v<= N/2 and u = kv with k in {1, -1}}.

Now, we will show that there exists a class of squares yielding non-trivial factorizations with identical distribution over the integers. We have as a theorem of abstract algebra that if N has at least two unique add prime factors, then there are at least two non-trivial square roots of 1 in Zn (not 1 or -1). Call these square roots r1 and r2. Notice that replacing 1 and -1 in the above set with r1 and r2 still yields pairs of congruent squares. But as r1 and r2 are non-trivial square roots, they are no congruent to 1 or -1. Thus the sets are disjoint, and the congruences from the 2nd set yield non-trivial factorizations. But both have identical distributions. Thus the probability that a randomly selected pair of congruent square yields a non-trivial factorizations is at least 0.5.

FOOT_NOTE (This is only a proof about the distribution of squares yielding trivial and non-trivial factorizations. It says nothing about the squares that we find in the QS. The QS is not a random algorithm, and its generation of squares is not random. Thus this is not a rigorous proof of the probability of success in the QS. However, heuristically, the result seems to hold.)

Thus, if we can generate p such pairs with some kind of statistical independence, we have a probability of at least 1 – 2^(-p) of success. As p grows, this quickly becomes sufficient.

But how do we generate congruent squares? Kraitchik's Method works by generating many relations of the form x^2 = y (mod N). This can be done by taking yi = xi^2 – N, as xi runs up from ciel(sqrt(N)). This works because x^2 as congruent to X^2 – N (mod N) for any x. Now, the set of squares is closed under multiplication. Thus, if we can find a subset of the y's with a square product, then those relations can be multiplied to give congruent squares. But how do we find a subset with a square product when there are clearly there are exponentially many to test. The answer involves finding the prime factorizations of the y's, specifically building vectors containing the exponents from their prime factorizations. Now, a number is square if and only if its exponent vector contains all even entries. Reduced modulo 2, this is the zero vector. When we need, then, is a subset of the exponent vectors modulo 2 with a zero sum. Modulo 2, the only scalars are 1 and 0. Therefore a subset sum is a liner combinations. We need a linear combination of vectors equal to the zero vector. In linear algebra, the set of such linear combinations over a matrix is called its nullspace. Recognizing this connection gives us two things: it gives us fast nullspace-finding algorithms, and it provides theorem telling when a non-trivial nullspace (containing more than the zero vector) will exist. Specifically, a matrix is guaranteed to be linearly dependent and have a non trivial nullspace if it has more columns than rows. The columns in our matrix correspond to the individual relations. The rows correspond to the unique prime factors of each value y. In order to calculate the number of relations needed to find a non trivial nullspace, then, we need to limit the number of unique prime factors of every relation. We achieve this by taking advantage of the fact that we can generate a surplus of relations, and only selecting those where the value y happens to be B-smooth for some value B (be the primes less than or equal to B). Then, as soon as we have found PI(B) +1 of them, their exponent vectors will produce a linearly dependent matrix. The nullspace can be found with Gaussian Elimination or one of several other more efficient algorithims, and it will give subsets of the original relations yielding congruent squares and factorization of N.

The Quadratic Sieve:

Everthing presented up to this point was known by 1925. What Carol Pomerence developed in 1981 was an efficient way to find smooth numbers in a polynomial progression. Pomerence realized that a classical Greek prime finding algorithm could be adapted to find smooth numbers. The algorithm, the Sieve of Eratosthenes, works by filling an array with flag values. Then, two is recognized as a prime, and all multiples of two are marked. The next unmarked number, 3, is prime, and all multiples of 3 are marked. The next unmarked index, 5, is prime. All multiples of 5 are marked. This process is continued for up to the square root of the length of the array. Then, all remaining unmarked numbers are prime.

The Sieve of Eratosthenes is a form of trial division. It finds primes by brute force division and removal of all non-primes. However, it is flawless trial division. The fact that we are working with all natural number up to some point allows us to predict exactly where each prime will divide the sequence. It is trial division without misses. This allows it to find all integers up to X in an incredible O(X log log X) time.

Pomerence recognized that the Sieve of Eratosthenes could be adapted to not only find prime numbers, but also to find smooth numbers. Rather than looking at the numbers in the sieve array that are unmarked at the end, we will look at the numbers that are marked. If we fill the sieve array with the actual numbers corresponding to each location, rather than simple flags, and divide each prime out of every location that it touches, then the set of B-smooth numbers will be exactly those locations that are reduced to one after all primes less then B have been processed. For this to work we have to sieve with higher prime powers as well, but this can be done without significant cost.

The sieve that we have so far described still only works for all natural numbers up to some point, while we need to find smooth numbers in a polynomial progression. But, it turns out, it can be easily adapted. What makes the sieve work is the fact that we can predict all locations where each prime will divide into the natural numbers. We need to be able to do the same for our subset of the natural numbers. To that end, let p be a prime dividing our progression. Then

p | (x^2 – N) <=> kp = x^2 -N <=> x^2 = N (mod p)

That is p divides the polynomial f(x) if and only if x is a square root of N (mod p). This immediately tell us two things: we only need to consider primes p for which the equation has a solution (primes under which N is a quadratic residue), and if N is a quadratic residue (mod p), then p divides f(x) for all x in the congruence classes of the two solutions to the equation x^2 = N(mod p).

These two observations allow the modified Sieve of Eratosthenes to be applied to our polynomial progression. This Quadratic Sieve finds all smooth numbers less than x in O(x log log x) time. This is sufficient to make Kraitchik's Method a sub exponential factoring algorithm.

Analysis and Optimal Parameter Selection

What is the runtime complexity of the Quadratic Sieve? On first encounter, it is not immediately apparent how the runtime depends on the size of N. It seems to depend much more on the smoothness bound B. Indeed, the runtime complexity almost seems constant! After all, we only need PI(B) + 1 B-smooth relations to factor. Why not take B=7? Then, we can factor with only PI(7) + 1 = 5 relations. This would, in fact, work. If we could find 5 7-smooth numbers in the polynomial progression modulo any number, we could factor that number. But the property of 7-smoothness is very rare. It is so rare, in fact, that it is often easier to find PI(10^9)+1 10^9-smooth numbers. Exactly how rare a property smoothness is in out polynomial progression, and hence exactly where the optimal smoothness bound falls, depends on the size of N. This is true because the starting magnitude and growth rate of our polynomial increase with N, and the probability that a number is smooth decays with its magnitude. The link between the magnitude of N and the runtime of the algorithm, then, is the selection of the parameter B. And to determine the Quadratic Sieve's asymptotic runtime complexity, we first need to investigate the optimal selection of B.

This is a difficult optimization problem. We need to express the time to factorization as a function of N and B, and then minimize with respect to B. The first thing that we need to perform this operation is some idea of how smooth numbers are distributed. Pomerence references work by Karl Dickman and Nicolaas Govert de Bruijn, giving $\psi(X, Y) \approx u^{-u}$ with u = ln X/ln Y, where $\psi(X, Y)$ is the smooth counting functions (the number of Y-smooth number less than or equal to X) [X]. Before we can use this estimate, however, we need to resolve two problems. First, the Dickman-de Bruijn only applies to the natural numbers. Can we use it to reason about our polynomial progression? And second, what is the value of X? What is the magnitude of the values in out polynomial progression? The second question has a simple answer. Pomerence reasons that if our optimal B turns out to be insignificant with respect to N, we can simply assume that the numbers in our progression have the same order of magnitude as their starting point (sqrt(N)). FOOT_NOTE(This reasoning is lent strength by the inclusion of -1 in the factor base and by the multiple polynomial optimization, allowing the polynomial values

to bracket N, not start at N, and allowing new polynomials to be used when values become too large. See chapter Algorithmic Optimizations for details.)

The second question is more troubling. Nothing is known about the distribution of smooth numbers on polynomial progressions. The QS is based in the assumption that the Dickman-de Bruijn function can be applied to our special subset of the natural numbers, and this assumption seems has been supported by the algorithm's success. In Pomerence's words: "The numbers we are trying to factor don't seem to mind our lack of rigor, they get factored anyway." [BOOK, 226].

With these two concerns out of the way, we can proceed. The excepted number of trials required to find one smooth number is u^u, the number of steps per trial is ln ln B, and we need about PI(B)/2 of them. Taking PI(X) = X/ln(X), this gives

$$T(B) = u^u (B/\ln B) \ln \ln B \text{ with } u = \ln N/2 \ln B$$

as an expression of the time to factor N with bound B. Taking the natural logarithm of this expression, finding its derivative, and solving for roots yields a minima at B = sqrt(exp(sqrt(ln N ln ln N))), indicating a runtime complexity of B^2 or exp(sqrt(ln N ln ln N)). These are somewhat approximant results, however. The many approximations made to arrive at these values lead Pomerence to qualify the optimal B value with a big-O.

FOOT_NOTE(This analysis does not consider the linear algebra step. However, that too can be done in O(B^2) time, and the efficiency of bit matrix operations is such that this step is dominated by the sieving.)

Algorithmic Optimizations

The Quadratic Sieve was the best known factoring algorithm for 15 years. As a result, it was the focus of much attention, and many improvements were made to the basic algorithm. In this chapter we present several algorithmic optimizations that we have implemented.

The Large Prime Optimization:

A composite integer must have a factor less than or equal to the ceiling of its square root. This fact allows us to extract a large number of "almost smooth" relations from the sieving process with no additional work. Specifically, if the value left in a location in our sieve array is less then B^2 after we have sieved out all primes less than B, we can deduce that the remaining number is prime, and thus that the corresponding relation is a B-smooth number times one large prime less than B^2. (It is important to note that this is not the same as being B^2-smooth.) The size of the space B^2 – B is such that many more almost smooth relations are found than smooth relations. The idea of the large prime optimization, then, is to attempt to put these relations use.

The idea is to find pairs of almost smooth relations that can be multiplied to yield usable values. Specifically, if two relations are found with the same large prime remainder, then they can be multiplied to yield a relation with a large square remainder. When we use relations in Kraitchik's Method , what we actually use are their exponent vectors modulo 2. And 2 modulo 2 is zero. Thus, these "almost smooth cycles" contain no non-zero entries in their exponent vectors beyond B, and can be treated as smooth relations in Kraitchik's Method. While it may seem a rather unlikely that two

almost smooth relations would share the same large prime factor, the great number of them found combined with the ”birthday paradox effect” (the effect that makes the odds in favor of at least two people in a group of 20 sharing a birthday) make it significant improvement to the QS. It is not without cost. The large number of almost smooth relations returned drastically increase bandwidth usage, and the usable cycles produces are composed of denser numbers (number with more factors) which de-optimize the matrix reduction step. In our experimentation, however, the large prime optimization more than doubled the speed of the QS.

The Multiple Polynomial Optimization:

There is a diminishing return in the sieving process. As the values in the polynomial progression grow, the probability that they are smooth decreases. The idea of the Multiple Polynomial Optimization is to fight this tendency by using multiple polynomials, and switching to a new one whenever the values in one progression become too large. Several versions of this idea were developed. We have implemented one due to Peter Montgomery.

The basic idea is to replace the x in our polynomial with a carefully selected linear function. Then, by modifying this linear function, we can generate multiple polynomials. If, in instead of $f(x) = x^2 - n$ we use the polynomial $f(x) = ax^2 + bx + c$ with $(b^2 – ac) = N$ then $af(x) = (ax + b)^2 \pmod N$. Relations generated with this polynomial still have the form of a square congruent to a quadratic residue, and as long as the value a is B-smooth (or a B-smooth number times a square) then we can sieve for smooth values in the progression f(x) and use the resulting relations $af(x) = (ax + b)^2 \pmod N$ just as we used the simpler relations. And how many of these polynomials can we construct?

An obvious question is how we will perform the sieving with this set of more complex polynomials. Luckily, it still holds that p divides f(x) if and only if x is in the congruence class of a root of the polynomial (mod p). And the set of primes for which the polynomial has roots are still those under which N is a quadratic residue. This is important, as it means that we can use many polynomials, without introducing new primes into the factor base and increasing the size of our matrix. The only change we need to make to the original algorithm is replacing the roots of $X^2 - N \pmod p$ with roots of $ax^2 + bx + c \pmod p$. Simple modular algebra gives these as

$r1 = (-b + t)a^{-1} \bmod p$

$r2 = (-b - t)a^{-1} \bmod p$,

where t is a roots of $x^2 - N \bmod p$.

The form of these roots hint at an efficient way to put this all together. The value t can clearly be precomputed for each prime in the factor base. Each individual polynomials is defined by two values a and b (for a given a and b only one c follows). And for each a, many b's will work. The value b only has to be a square root of N (mod a). And if a has k unique odd prime factors, then N has 2^k square roots (mod a). Thus, if we intentionally chose a with many unique factors, we can precalculate the value a^{-1} modulo each prime in the factor base, and use them for thousands of b’s.

A finally question is how we should select the value a. Montgomery first selects a range over which the polynomial will be sieved, and then selects a to minimize the absolute value over this interval. His result is a =

In order that N have square roots mod a, N must be a quadratic residue mod all factors of a. Thus the set of potential factors of a is exactly the primes in our factor base. But notice that during the sieving process each prime in the factor base must be invertible mod a. This is a

contradiction. It is solved by temporarily removing those primes that compose a given a from the factor base. This is a cost. We will miss some smooth numbers as we sieve with the family of polynomials from any a. We can balance it, however, by by not selecting too many primes for inclusion in a. And the speed up from the Multiple Polynomial Optimization dwarfs this cost.

FOOT_NOTE (This idea of taking a with many unique prime factors and pre-calculation much of the roots was actually developed by Pomerence as an improvement to Montgomery's Multiple Polynomial Quadratic Sieve, and is known in the literature, somewhat inaccurately, as the Self-Initializing Multiple Polynomial Quadratic Sieve).

Negative One Optimization

The final algorithmic optimization that we implemented was simply to include the number negative one as a "prime" in our factor base. This allows negative number to be seen as smooth, and makes it possible to center the range of values sieved around 0, rather than having to start at 0, which double the quantity of numbers at any one magnitude and helps fight the growth of the polynomial. The inclusion of negative one is possible because it can be treated like any other prime in the matrix step (although obviously it must be a special case in the actual divisibility testing). If we add a column to the matrix corresponding to negative one, then any subset sum with a zero in this location is positive, and might be a square. This exactly corresponds to the requirements for other primes, and allows the Negative One Optimization to be used with minimal change to the algorithm and the infinitesimal cost of adding one column to our matrix.

Java Parallel Multiple Polynomial Quadratic Sieve

In order to achieve our research goals, we implemented the Quadratic Sieve with the algorithmic optimization described in the previous chapter. We named our system JPMPQS. This chapter describes the design and implementation of JPMPQS.

Specification

The basic function of JPMPQS is to factor integers in parallel over multiple machines. Thus the most central requirement is that it consist of some sort of network of remote machines, and that it take composite input and output a list of factors. Beyond this, however, we developed the following requirements for the system.

- Read job request from an XML file, allowing data-driven configuration of key parameter to simplify testing.
- Provide detailed feedback in XML job reports containing breakdown of time in each step of the algorithm, to help with analysis and parameter tuning.
- Verify all data from remote clients, so that subverted clients cannot introduce error
- Log all data, and allow jobs to be resumed (after being stopped or a failure of any part of the system)

Design

Our design for JPMPQS is composed of three main systems: a job manager, a sieving client, and a matrix processor. Each of these components is a separate application written in Java (with the partial exception of the matrix processor, which uses C code), and they communicate with each other via TCP sockets. To factor a number, the job manager will first process a job request from the user. It will then dispatch messages to the sieves to tell them to start sieving. They will sieve for smooth numbers, requesting new polynomials (or a base to generate a family of polynomials) when necessary, and sending them to the job manager. The job manager will send these numbers (packaged in relations with the input to the polynomial that produced each number) to the matrix processor. When the matrix processor finds a linear dependency, it computes the resulting factorization, and, if it is non-trivial, sends it back to the project manager. This basic structure is shown in the following diagram.

In each component, the network system is separated from the QS system by way of abstract interfaces. Each local QS component maintains abstract interface representing the other remote components, and interacts with this interface with no knowledge of their function. Concrete network classes then implement these interfaces to communicate over TCP sockets with the remote machines. The network components are standard Java client/server applications making heavy use of the state design pattern to reflect protocol states, and this document will not address them in more depth.

Job Manager:

The job manager's function is to interact with the user and manage the rest of the system. It can be seen as a master entity, while the other components are slaves. It takes job requests from the user, and communicates with the other components to get the job done. Beyond verifying returned results and shuttling data, the job manager is not involved in the actual business of factoring. Key functions of the job manager include managing clients, interacting with the user and parsing job XML request, varying all data returned from clients, maintaining a database of almost smooth relations and looking for usable cycles, keeping track of the number of relations found and deciding when to try to reduce the matrix, and finally writing the XML job report log.

Sieve Client:

The sieving client sieves a polynomial sequence for smooth numbers. When notified of a job by the job manager, it requests a polynomial base, and then proceeds to generate an extended factor base for that polynomial base (extended in that it contains the primes in the factor base, as well as precomputed logarithms and roots for the sieving), and sieve successive intervals in the polynomials that it can generate, passing the found smooth numbers back to the job manager. The sieving client is a slave. It starts when told, pauses when told and stops when told. The sieving itself is carried out using subtraction of logarithms in place of division of large integers. Low precision base-2 logarithms, generated rapidly by counting binary bits, are used for all logarithms computed in the fly. The use of approximate logarithms makes the sieving process approximate. It is a initial filter (sieve) which discards the vast majority of candidates. The few number that pass through the sieve are than verified by way of traditional trial division.

Matrix Processor:

The matrix manager receives smooth relations from the job manager, puts the exponent vectors mod 2 of their prime factorizations into a gigantic bit matrix, and looks for elements of its nullspace. As the final receptacle for relations, and so has to find and remove duplicate relations that will water down the usable nullspace elements in the matrix. The actual matrix reduction we did not implement our selves. Rather, we used a Block-Lanczos routine from FLINT (Fast Library for Number Theory) by William Hart and Jason Papadopoulos, called via JNI.

Implementation

One focus of the project was top explore the process of taking theoretical work and implementing it in Java. This process was more difficult than we had anticipated. There were two main difficulties. First, we had research and implement many other algorithms used as part of the Quadratic Sieve. Particularly, the Java BigInteger class only provides basic functionality, and we had to extend it to calculate N^{th} roots and logarithms, as well as modular square roots by way of Shanks-Tonelli algorithm, Hensel lifting and the Chinese Remainder Theorem. Writing optimized implementations of these algorithms was a difficult task, particularly Hensel lifting, and developing our mathematical library actually occupied almost one third of our total development time.

The second primary hurdle was coming up with implementation-level detail of the QS itself. Examples include how to adjust precalculated sieve roots to allow sieving in discreet blocks, how to best generate polynomial bases for use by sieve client, and how to actually find a factorization given an element of our matrix's nullspace. These were not terribly difficult problems, but they have some subtlety and took us a number of days to solve. And they show the detail-gap that we had to fill in order to develop JPMPQS. As an example of the , we here present the 3rd problem and put solution in greater depth.

The matrix reduction step in the QS gives us a subset of the relations that, multiplied together, will yield a pair of congruent squares. All we have to do to factor, then, is multiply them, take their square roots, and calculate the greatest common dominator of their sum and N. This seems simple enough, and is given very little attention in the literature. However, in practice it is not simple. We cannot just multiply the relations, because the resulting numbers will be of unmanageable magnitude N^B. Now, we do not actually need these numbers. We only need their square roots reduced modulo N. The solution to the problem, then, is to take advantage of the fact that we know the prime factorizations of each relation. We can thus generate the prime factorizations of the squares and of their square roots, and then calculate the those roots modulo N by way of fast modular exponentiation. (Astute readers will note that the large prime optimization actually makes this more complicated, as the resulting large square factor are not represented in our exponent vectors. We solve this problem by refactoring each large-prime relation into a larger sparsely encoded exponent vector.)

A final difficulty in the implementation of JPMPQS was he software engineering. The final system consists of three separate systems in four Java packages and a C module containing over 40 files. We did out best to employ modular black-box design and use abstract interfaces to partition the complexity into manageable units, and for the most part this reductionism worked. But JPMPQS it is one system, and at times we had to view it as such. Thread, process and system synchronization was the area where this particularly came up. To make this work we had to strictly separate design from implementation. We had to careful diagramming a synchronization plan for the entire system avoiding race conditions, and then refer to it constantly when implementing individual components.

Results

Using this implementation of JPMPQS, we then proceeded to gather data.
All tests described here were performed in the Computer Science lab at SUNY Potsdam, on 14 IBM workstations running Linux

Parameter Selection:

Our first goal was to investigate optimal parameter selection, so that all additional data might be gathered with optimal parameterization. Off the multitude of parameters to JPMPQS, we chose to investigate three which we considered most central to the algorithm, and which we thought might be affected by our parallelization and algorithmic optimization. These thee parameters were sieve smoothness bound, sieve interval, and sieve cutoff.

The sieve cutoff is the value below which an entry in the sieve array must be reduced during the sieving for it to be checked rigorously checked for smoothness with trial division. This parameter controls the link between the fast approximate sieve, and the trial division that makes sure that that only smooth (or almost smooth in the case of the Large Prime Optimization) values are returned to the job manager. If the sieve functioned perfectly, the sieve cutoff could be zero. The use of approximate logarithms and the exclusion of 2 from the factor base (excluded because its evenness complicates the calculation of modular square roots), however, necessitates a significantly higher value. Testing during the development of the sieve client indicated that a value in the range of 15 maximized smooth number return, irrespective of the value of N. In JPMPQS, however, the Large Prime Optimization makes use of almost smooth numbers. Thus we expected to optimal value in JPMPQS to depend on N (or B). Specifically, the theoretical value to catch all almost smooth values is 2*log2(B), and we expected the optimal value to be a multiple of this.

Testing XXXXXXX decimal digit semiprimes with a range of sieve cutoffs produced the following results.

Notable in these results is the constant nature of the optimal value. For all three classes of numbers, a sieve cutoff in the range of 25 was optimal. This contradicts our expectation. When N is large, a higher cutoff would without question increase the number of almost smooth cycles found. However, this advantage seems to be dwarfed by the constant trial division cost of a larger cutoff.

The interval is the interval over which each polynomial is sieved before switching to a new one in the Multiple Polynomial optimization. The goal in selecting this parameter is to balance the gain only sieving on average smaller values with the initialization cost of switching polynomials. Each prime in the factor base divides into the sieve array at intervals equal to its size. Thus a sieve interval less than B would exclude some primes in the factor base. For this reason, we took B as our minimal sieve interval, and investigated multiples of this value. Tests yielded the following results.

These are the least crisp of our results. Particularly for smaller values of N, there is significant variation over our test range.

The smoothness bound is the most important parameter to the QS, and the key to its runtime complexity. Following O(sqrt(exp(sqrt(log n log log n))))

Parallelization:

Our next aim was to gather data on the efficiency of the sieve parallization. To investigate the parallization, we factored the same 60 digit semiprime repeatedly, using from 1 to 10 sieving nodes, and compared the resulting times. We were particularly interested in the toal time (sum of time spent on ever machine). With a perfect parallization, the total time should remain constant regardless of the number of machines used, while a less perfect parallization should be marked by an increase in total time as the number of node rises.

The data follows.

Elapsed time decreases with node count, as expected, but ff note in this table is the matching decrees in total time. This occurs because, while the sieve are running, the job manager and matrix processor are not necessary fully utilized. To check the parallization we need to look only at the total sieve time (time spent in the sieve nodes). Here, we have remarkable data. Notony is the sieve time nearly constant, it hints at a downward trend. The data hints at a sieve synergy, whereby more sieves factor in less total time. This is, of course, not possible. The slight trend is either an anomaly in the data, or the result of the fact that with more sieve the factorization could be finished using only initial polynomial assignments and not having to wait for new ones.

Runtime Complexity:

Our final result that we sought was to document the empirical runtime complexity of JPMPQS, and compare it to Pomerence's theoretical complexity. Big-O analysis is asymptotic, so it would in no way challenge Pomerence of our data did not fit. But I good fit would support

the theoretical analysis, and would support the use of the theoretical function to calculate expected factoring times for large numbers. To evaluate runtime complexity, we factored 10 semiprimes with from 40 to 85 digits. For consistency in magnitude, all were chosen with leading digit 8.
Factorization times for these numbers, graphed against Pomerence's theoretical complexity function scaled by 10^(-9) for best fit, follow.

This graph

These results are preliminary. They come from relatively few tests run on relatively few machines, and fall short of statistically significant results. This was primarily a result of the resources available to us. The computers on which they were run were in use during the day by introductory programming classes, limiting us to test cases which could finish in a single night.

Conclusion

What the Quadratic Sieve Is:

The quadratic

What the quadratic Sieve is Not:

The quadratic sieve is not an entirely rigorous algorithm. It contains a random step (taking modular square roots) and its runtime analysis is probalistic, and based on unproved assumptions about the distribution of smooth numbers in polynomial progressions and the statistical independence of squares found by the combination of these smooth numbers. Still, it was in practice the best known general factoring method until 1996, and remains relevant for number of fewer than 110 digits.

We have implemented the Quadratic Sieve in Java.

What we will do in the future

100 Quadrillion Calculations Later, Eureka! – NYT article to cite