A Parallel Implementation of the Quadratic Sieve Factoring Algorithm in Java

By Ammon Bartram
SUNY Potsdam
bartraah190@potsdam.edu
Adviser: Brian Ladd
laddbc@potsdam.edu

Potsdam
THE STATE UNIVERSITY OF NEW YORK
1816

Introduction

Reducing a composite integer to its prime factors is a difficult task. Indeed, the lack of a polynomial time solution to the factoring problem is the basis of the RSA cryptographic system, one of the most popular public-key systems in use today. The Quadratic Sieve factoring algorithm, developed by Carl Pomerance in 1981, was the first clearly sub-exponential time factoring algorithm known, and is still the algorithm of choice for integers under 110 decimal digits. Pomerance is a mathematician, however, and his work on the Quadratic Sieve (QS) is theoretical and lacks empirical data. The aim of our research, then, is to gather data supporting Pomerance's analysis, giving special focus to the effects of parallelizing the algorithm.

Congruent Squares

The QS is best understood as an extension of Fermat's factoring method. Fermat attempts to express n (the number to be factored) as a difference of squares. Then

$$n = (u^2 - v^2) = (u - v)(u + v)$$

gives a factorization. Finding such squares, unfortunately, is no more efficient than factoring by trial division. In 1925, however, Belgian mathematician Maurice Kraitchik realized that it is sometimes sufficient to find the weaker condition

$$(u^2 - v^2) = kn, \text{ or } u^2 \equiv v^2 \pmod{n}$$

Then, there can be shown to be a probability of at least 0.5 that $gcd(u-v, n)$ is a non-trivial factor of n. If we can generate a number of pairs of such congruent squares, then probabilistically we will be able to split n .

Exponent Vectors

The QS finds congruent squares by generating many relations of the form $x^2 \equiv y \pmod{n}$ by taking $y_i = x_i^2 - n$ as x_i runs up from the square root of n ($x^2 \equiv x^2 - n \pmod{n}$ for any x). Then, if a subset of the y's can be found with a square product, those relations can be multiplied together to yield a pair of congruent squares. Such a square subset can be found by applying an algorithm from linear algebra (nullspace finding) to the vectors containing the exponents from the prime factorizations of the y's. To make this possible, we need to limit the size of these vectors (limit the number of unique prime factors of each y_i). To this end, we take advantage of the fact that we can generate a surplus of relations, and select only those where the value y happens to be a product of relatively small primes (this quality is known as smoothness).

The Quadratic Sieve

Smooth values are located in the polynomial progression by taking advantage of the fact that we can predict exactly where each prime will divide the sequence. Specifically, the members of the residue classes of the two solutions to the equation $x^2 \equiv n \pmod{p}$ are the locations where each prime p divides the polynomial. By only dividing at these location, we can avoid most of the cost of trial division, and find smooth values very efficiently. This technique is known as sieving, and gives the Quadratic Sieve its name.

Multiple Polynomials

As the value of an integer x increases, the probability that x is smooth decreases. This leads to a diminishing return in the sieving process as the value of the polynomial grows. To fight this, we replace x in the polynomial with the linear function $ax + b$ (a and b must satisfy several constraints for this to work). This allows many pairs a and b to be used, yielding many unique polynomials and allowing the sieve to start over with a new polynomial whenever the value of one polynomial becomes too large.

Parallelization

Sieving for smooth numbers in the polynomial progression is the time-dominant step in the QS algorithm, and this sieving can be carried out in parallel. The multiple polynomial optimization, in particular, makes this efficient. Each sieving node can be assigned a unique polynomial base (the value a in the preceding paragraph), and can independently generate thousands of b values. This allows each sieve to operate independently, and achieves a near perfect division of the sieving across multiple nodes.

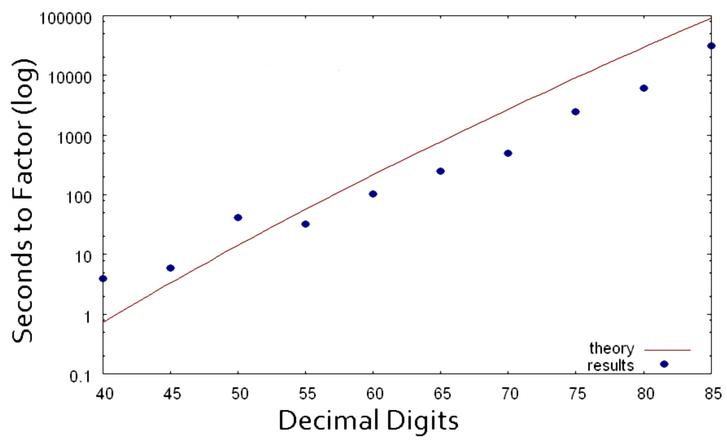
Implementation

We implemented the Multiple Polynomial Quadratic Sieve in Java, using a master-slave architecture. A master job manager takes job requests from the user, and distributes polynomial bases to multiple remote slave sieve clients. These sieve clients then generate polynomials, find smooth relations, and return them to the manager. When enough have been gathered, the job manager calculates exponent vectors, stores them in a matrix, finds its nullspace, and finally factors n. The sieving itself we optimize using subtraction of low-precision logarithms in place of division. The matrix reduction we perform with a C Block-Lanczos routine (from the FLINT number theory library) called via the Java Native Interface.

Results

Our first aim was to gather data on the optimal selection of the smoothness bound (the maximum prime factor that a number can have to be considered smooth). To this end, we factored semiprimes (the product of two primes, and provably the hardest case of the factoring problem) with from 30 to 70 decimal digits, using a range of smoothness bounds. In all cases, we found the optimal bound to be about one tenth the theoretical optimal for the single polynomial QS. This supports Pomerance's hypothesis that the multiple polynomial optimization would reduce the optimal smoothness bound.

Our second aim was to provided data supporting Pomerance's runtime analysis. To that end, we factored semiprimes with from 40 to 85 decimal digits (the largest number factored is shown in the left margin), and plotted them against Pomerance's predicted complexity function (scaled to best fit, as allowed in big-O analysis).



This data strongly supports Pomerance's analysis of the Quadratic Sieve. Our data falls plausibly along his complexity function, and illustrates the sub-exponential time of the Quadratic Sieve by its apparent sub-linear trend against a logarithmic y-axis.

852852807586981103031574790855087845313917281711416459802784148066838428013507828502718697 * 74703789293682328078978027974715437812230077