Parallel Quadratic Number Field Sieve

Daniel Ron

December 20, 2011

Abstract

The Quadratic Number Field Sieve (QS) provides a fast and easy to implement method to factor a number. QS works by setting up a congruence of squares modulo n, which gives us a factor of n roughly 2/3 of the time. I set out to implement QS in parallel in MIT Cilk using the Gnu Multiple Precision Library. I succeeded in improving the performance of two functions in QS through parallelization. My results agreed with previously shown performance. Specifically, the data collection phase of QS is parallelized better than the data processing phase[3]. An efficient way of handling the data processing phase remains to be shown.

1 Introduction

The problem of factoring large numbers has been the basis of modern security for years. Most cryptography algorithms rely on the difficulty of factoring large semi-primes, that is numbers of the form $N = pq$ where p and q are both large primes of similar size. Attempts to factor large numbers are largely for research purposes, as it helps give us an idea of what size numbers are "safe" from attacks. The current accepted standard is that N must be about 2048 bits long to be considered secure [1]. This number, of course, grows as computers become more powerful, however it is still believed that 4096 bits will remain secure for a very long time.

The Quadratic Number Field Sieve (QS) was discovered in 1981, and was the asymptotically fastest known algorithm for factoring integers up until the discovery of the General Number Field Sieve (NFS) in 1996. During its reign as the fastest factoring algorithm, it managed to set several records, including finding the factorization of the 129-digit number, RSA-129. QS is still of interest because it is much easier to understand and implement than NFS, so it's an ideal algorithm for a hobbyist to attempt to implement and optimize. However, in practice when groups are actually attempting to set records, they use the NFS.

QS works based off of congruence of squares modulo an integer n. If we can find a pair of integers a, b such that $a^2 \equiv b^2 \mod n$, then with about 2/3 probability $\gcd(|a - b|, n)$ will be a nontrivial factor of n. QS works in two stages to find such pairs, the data collection stage, and the data processing stage. In the data collection stage, we build a very large sparse matrix over F_2. In the data processing stage, we find the right nullspace of the matrix. The nullspace can be used to find a pair a, b. The first step can be parallelized fairly trivially, but the data processing stage is significantly harder to parallelize, because most algorithms require that each node have access to the entire array, which grows to be quite large.

1.1 Motivation

I was interested in attempting a numerics problem using C or C++ to see what kind of speed I could get based on my own algorithms. I went in knowing that it would be unlikely to make any significant improvements over previously written QS implementations, so my original goal was to optimize the functions that make up QS as much as possible. There appeared to be several opportunities for paralleling, and many tuning parameters that could be optimized, so it thought it would be an ideal problem to work on. Furthermore, this was my first time ever doing any actual coding for parallel computing, and only having to worry about embarrassingly parallel problems seemed like a good idea.

1.2 Implementation Details

I was interested in using MIT Cilk from the start, because it's based on C, and made parallelizing relatively simple. Most parallelization was done through parallel for loops done by recursively dividing the range in half, and spawning recursively on the two halves. All benchmarking was run locally on a 4-core Intel i5 750 running at 3 GHz on each core, and 4 GB of RAM.

2 Algorithm

In the data collection stage, QS collects a set of a_i, b_i such that $a_i^2 \equiv b_i \mod n$. In the data processing stage, we try to find some set of indices, $\{s_1, s_2, ..., s_l\}$, such that

$$\Pi_{i=1}^{l} b_{s_i} \equiv b^2 \mod n$$

for some b. Likewise, if we have

$$\Pi_{i=1}^{l} a_{s_i} \equiv a \mod n$$

then we'll have found a congruence modulo n:

$$a^2 \equiv b^2 \mod n$$

To efficiently find the right set, we look at the prime power representation of each b_i, such that we have $b_i = p_0^{e_{i0}} p_1^{e_{i1}} ... p_m^{e_{im}}$ For squares, each exponent e_j will be even, thus we want to find the set of b_i who's product will have even exponent powers. To simplify this search, we can consider all exponents mod 2, so we're looking for exponents to all be 0. Now if we bound the size of the largest prime in the prime power representation, we can represent each b_i as a vector $(e_{i0}, e_{i1}, ..., e_{im})$, where multiplication of two b_i corresponds to adding the vectors. So if we collect a large amount of b_i, we want to find a set who's vector sum is 0. If we store the vectors as column vectors in a large matrix, then finding the correct subset is exactly the problem of finding the right nullspace.

One of the biggest tricks as to why this algorithm works well is bounding the largest prime, p_m. A number who has no prime factor greater than B is called B-smooth. In using smooth numbers, we first of all determine the size of our system, which makes handing all the data easier. But furthermore, it means we can build up our set of b_i in reverse, by selecting numbers who are products of small primes, as opposed to factoring each b_i. Our collection of primes are called our "factor base".

The basic idea is as follows. In practice, there are many optimizations that make this work. We start with an interval of $a_i^2 \equiv b_i$ to test. For each prime, p_j, in our factor base, we divide each b_i by p_j and store the result. After going through our entire factor base, we look at each value in our interval that is 1, and those will be our B-smooth numbers. We then use these to build up our matrix and find the nullspace.

In practice, of course, this would be highly inefficient. The first performance trick is generating the a_i via a polynomial. We use a polynomial of the form $f(x) = (Ax + B)^2 - n$. Note that $f(x) \mod n \equiv (Ax + B)^2 \mod n$, so therefore we have $a_i = Ax + B$, and $b_i = f(x)$. If we solve $f(x) \equiv 0 \mod p$ for prime p, then we'll find a b_i that p is a factor of. But more importantly, notice that for a given root α, that $f(\alpha + kp) \equiv 0 \mod p$ for integer k. So we can find all b_i that are multiple of p with one calculation. Actually dividing each number would be far too expensive, so instead we subtract (or equivalently, add) logarithms of p_j for each $f(\alpha + kp)$. Thus we can start each value in our range at 0, and add up $\log p_j$ until the sum is equal to $\log b_i$. Another trick we can do is to use the integer logarithm, which is much faster to calculate, but obviously less accurate. This introduces additional errors into the system, but dealing with the errors is easier than doing the floating point operations.

2.1 Approach

My approach was to look at the problems and functions that are used frequently in QS and see what improvements I could make. I came up with five problems that came up frequently in QS: primality checking, modular square root, GCD, factoring "small" numbers, and calculating the nullspace over the finite field F_2. These all come up frequently in performing QS, and my thought was that if I could improve upon them, I could improve the performance of QS as a whole. In doing research, I found that the modular square root and GCD algorithms that work best on this data don't parallelize well. However, the algorithms for primality testing and factoring relatively small numbers could be parallelized easily. It wasn't clear if calculating the nullspace was possible to parallelize. I thus focused my efforts on primality checking and small number factoring, and then began work on calculating the nullspace.

2.2 Primality Checking

Primality checking is the problem of determining if a number is prime. Naively, one could do this by trial division, but that would take $O(\sqrt{n})$, which can be very very bad for large n. MR runs in $O(k \log^3 n)$, where k is a tunable parameter that affects accuracy. MR could actually be called a "composite checker," because it checks if a number is composite, and returns prime if it can't prove it is composite. The basis of MR comes from Fermat's little theorem. We start by randomly choosing an integer a. If n is prime, then one of the following must be true:

$$a^d \equiv 1 \mod n \quad \text{for odd } d$$

or

$$a^{2^r d} \equiv -1 \mod n \quad \text{for some } r \text{ such that } 2^r d \leq n-1$$

Because there are finitely many equivalences to check, we can just run through all of them. If we satisfy either of the conditions, then we can return that the number is prime. If after going through all possible r we haven't satisfied either of these, then we return composite. However, this test isn't 100% accurate in this state. There exists combinations of numbers, (a, n), where n is composite, that will result in the algorithm returning prime just by bad luck. For a randomly chosen a, this happens about $1/4$ of the time[4]. To fix this, we choose multiple a's, increasing the probability that our result is correct with each additional iteration. For k iterations, we have $1 - 4^{-k}$ accuracy.

Given an a, all the calculations are almost entirely serial to the point that parallelizing would decrease performance due to overhead. However, each one of these serial problems can be run in parallel over each core using a different a on each core. Furthermore, if we ever find an a that confirms that n is composite, called a "witness" for n, then we can abort all threads and return composite.

Primality checking is more important in finding all the factors of a random number, instead of attempting to factor a chosen large semi-prime. This is because when factoring a semi-prime, we know we're done after finding one of the two factors, however we aren't this lucky when factoring a random integer.

2.3 Small Number Factoring

There are a few points QS where it is beneficial to be able to factor small or smooth numbers quickly. The way this is used changes depending on how we implement QS. For example, as we attempt to find a set of smooth b_i, we can keep track of the factors of every b_i stored in an auxiliary array. However, for factoring large numbers this can waste gigabytes of space, which will slow us down with more reading and writing to disc. Thus we can ignore storing factors, and factor the b_i after we've collected them. There were two methods I considered for small number factoring: trial division, and the Pollard-Rho method. I focused on the Pollard-Rho method, as it would be the most versatile, and the most interesting to work with.

2.3.1 Pollard-Rho

Pollard-Rho is a factoring algorithm that's based on cycle detection with a pseudo-random sequence. In factoring n, we define the function $f(x) = x^2 + a \mod n$ for some a. We initialize with $x = y = 2, a = \text{randint}$. In each step we do

$$x = f(x),\ y = f(f(y)),\ d = GCD(|x - y|, n)$$

If $d \neq 1$, then we may have found a nontrivial factor of n. d could be n though, in which case we return failure. We can run many threads of this at once, each seeded with a different a, and abort as soon as one returns a positive factor.

Pollard-Rho has poor performance on large semi-primes, having a theorized runtime of $O(\sqrt[4]{n} \text{ polylog } n)$. However, for smooth numbers, we're expected to find a factor p of n in $O(\sqrt{p})$. So the smaller the factors, the faster we're going to find them. The runtime is still dominated by the largest factor of n, but if n is smooth with a relatively low bound, then we know that we'll have fast factoring.

2.3.2 Trial Division

Trial Division only works on *very* small smooth numbers, as its running time is bound by $O(p) \approx O(\sqrt{n})$[2]. Furthermore, each step is only slightly cheaper than Pollard-Rho, so its constant factor doesn't help that much.

3 Results

There was a lot left to be done to have a properly functioning Quadratic Number Field Sieve. However, there were improvements seen from the initial stages of parallelization.

3.1 Miller-Rabin Primality Testing

Miller-Rabin was fairly trivial to parallelize, requiring only a parallel for loop to split the threads up over the cores. I benchmarked my primality tester against the Gnu Multiple Precision Library's (GMP) built in primality tester, which was also based off of Miller-Rabin. I found that for larger numbers, my tester ran faster on just one core, and ran significantly faster when run over over multiple cores. The following graph shows the runtimes of the primality testers in determining that a large semi-prime is prime, which is computationally the hardest thing for the algorithm to do.

A speedup of over one order of magnitude was achieved with only four cores for larger numbers. All the methods worked at about the same speeds for numbers around 100 digits, and GMP's built in primality tester ran significantly faster on smaller numbers. GMP's built in function most likely uses a different method for smaller numbers, as Miller-Rabin can be a bit overkill due to the overhead required. Such cutoffs are usually very system dependent, and change as machines get more and more powerful. Sufficient

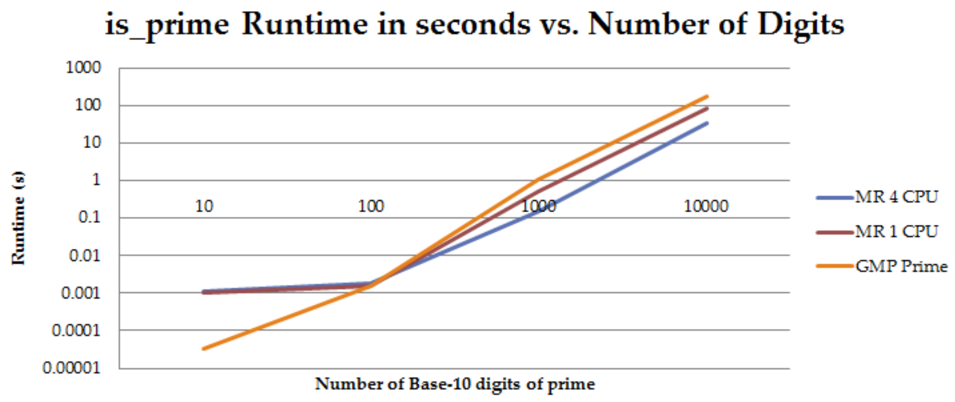

Figure 1: Primality testing on large semi-primes. Runtime vs. number of digits

testing would have to be done on each system to calibrate cutoff values. Ideally such a function will be built in future iterations, so that one doesn't have to repeatedly guess-and-check manually. The only cutoff performance tweak done is checking if n is divisible by 2, 3, 5, 7, or 11 before doing anything else. For testing a random number, this accounts for more than 2/3 of all possible n, so will be able to quickly catch those n.

These results aren't amazing, but were still very exciting. They showed how easy it was to see speedup from parallelization. Furthermore, I have yet to attempt higher levels of optimization beyond high-level concerns about runtime, which leaves a lot of room for low-level tweaks to improve performance.

3.2 Pollard-Rho Factoring

Pollard-Rho performed as expected. It performed significantly faster than trial division on anything besides very small numbers. It even did a decent job at factoring small semi-primes. For example, it managed to factor $10000001004200000030117 = 100000000003 * 100000010039$ in about 2s on 1 core, and about 100 ms on 4 cores. This isn't considered fast by modern standards, but it was significantly faster than trial division, which ran for several minutes before seg-faulting. As such, it was able to quickly factor randomly generated smooth numbers with a smoothness bound of about 1000000.

Like the Miller-Rabin test, I did not perform any low-level performance tweaks. There is likely much room for improvement, and hopefully some improvements will be made in the future.

4 Discussion and Moving Forwards

Clearly, there is a lot more work needed to be done to have a functional QS implementation. The biggest hurdle is handling the linear algebra on the final matrix. The matrix we're working with can be very large, on the order of many gigabytes, so performing operations on the whole thing isn't feasible on anything short of a supercomputer. Furthermore, the Block Lanczos algorithm, which is the fastest known algorithm for this data type, really requires shared memory across all the nodes. So distributed algorithms will be very difficult to implement. For small numbers, we can safely just store our array in memory, but as reading and writing to disc become an issue, runtimes increase exponentially. I would like to spend more time researching how parallel Block Lanczos can be safely implemented.

There are also faster, but more complicated methods for factoring smooth numbers such as the Elliptic Curve Method (ECM). ECM has the benefit that even if the number has a prime factor or two that are above our smoothness bound, ECM will still be able to quickly factor them. However, ECM is significantly more complex than Pollard-Rho, and as such would be that much more difficult to parallelize and optimize.

As I mentioned in the results, there remains many places where cutoffs could be found to optimize performance. For example, I found that below about 100 digits the serial primality tester built into the GMP library seemed to work faster the my parallel implementation. This is especially true with the various factoring methods, as different factoring algorithms work better on all sorts of different kinds of numbers. Pollard-Rho is a nice, fast, factorization algorithm, but works better on "small" numbers. ECM similarly works very quickly, and can deal with numbers that have a few large prime factors. However, my research hasn't revealed anything about being able to tell how large the largest prime will be, so there may not be a way to "analyze" the number before factoring. However, there are still other places where some theoretical and empirical analysis could really help performance. For example, we randomly choose seeds for Miller-Rabin, Pollard-Rho, *and* multiple polynomial QS. It may be best to leave them as randomly chosen to keep optimal expected runtime, but it remains to be actually attempted. I believe that with some research, improvements into how these numbers randomly chosen.

There are still a large number of places to look for optimization, many of which can't be tested until I have a complete running implementation of QS. My goal is to get a serial version of Block Lanczos working so I can begin testing potential optimizations.

References

[1] Burt Kaliski. How these discoveries affect rsa security's products: Frequently asked questions. Technical report, RSA Laboratories.

[2] Brandon Luders. An analysis of the complexity of the pollard rho factorization method. 2005.

[3] Peter L Montgomery. A block lanczos algorithm for finding dependencies over gf(2). *Lecture Notes in Computer Science*, pages 106–120, 1995.

[4] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

```
//Miller.cilk

#include <cilk-lib.cilkh>
#include <stdlib.h>
#include <stdio.h>
#include <gmp.h>
#include <time.h>

#define GRANULARITY 2

/*
 * The heavy lifting. Returns 1 if it thinks n is prime, 0 if n is composite
 */
cilk int miller_rabin_aux(mpz_t a_base, mpz_t n)
{
  int i=0, s=0, prime=1; // s is proportional to log_2(n)
  mpz_t a, d, n_minus_one;
  mpz_init(n_minus_one);
  mpz_init(a);
  mpz_init(d);
  mpz_sub_ui(n_minus_one, n, 1);
  mpz_set(d,n_minus_one);

  while(mpz_even_p(d))
  {
    mpz_fdiv_q_2exp(d,d,1);
    s++;
  }

  mpz_powm(a,a_base,d,n);

  if (mpz_cmp_ui(a,1) == 0)
  {
    goto quit;
  }
  for (i; i<s-1; i++)
  {
    if (mpz_cmp(a,n_minus_one)==0)
    {
      goto quit;
    }
    mpz_powm_ui(a,a,2,n);
  }
  if (mpz_cmp(a, n_minus_one) == 0)
  {
    goto quit;
  }
  prime = 0;
  quit:
    mpz_clear(a);
    mpz_clear(d);
    mpz_clear(n_minus_one);
    return (prime);
}

/*
 * Parallel for loop. Uses divide and conquer with granularity
 */
cilk int cilk_for(int lo, int hi, gmp_randstate_t state, mpz_t n)
{

  int mid, up, down, i, prime;
  mpz_t a;
  mpz_init(a);
  if(hi-lo>GRANULARITY)
  {
    mid = (lo+hi)/2;
    down = spawn cilk_for(lo, mid, state, n);
    up = spawn cilk_for(mid, hi, state, n);
    sync;
    return (up+down)/2;
  }
  for (i = lo; i<hi; i++)
  {
    mpz_urandomm(a, state, n);
    prime = spawn miller_rabin_aux(a,n);
    sync;
    if(prime == 0)
    {
      mpz_clear(a);
      abort;
      return(0);
    }
  }
  mpz_clear(a);
  return (1);
}

/*
 * Master function. Runs k iterations of miller_rabin_aux
 * Returns Prime (1) if every iteration returns prime
 * Returns Composite (0) if any iteration returns composite
 */
cilk int miller_rabin(mpz_t n, int k)
{
  int prime;
  gmp_randstate_t state;
  gmp_randinit_default(state);
  gmp_randseed_ui(state, time(NULL));

  if (mpz_fdiv_ui(n, 2)==0 || mpz_fdiv_ui(n, 3)==0 || mpz_fdiv_ui(n, 5) == 0
|| mpz_fdiv_ui(n, 7)==0)
  {
    return (0);
  }
  prime = spawn cilk_for(0,k,state,n);

  sync;
  return (prime);
}


cilk int main(int argc, char *argv[])
{
  mpz_t n;

  int k=10,prime;

  mpz_init_set_ui(n, 100019);
  if (argc > 2)
  {
    mpz_set_str(n, argv[1], 10);
    k = atoi(argv[2]);
  }
  prime = spawn miller_rabin(n,k);
  sync;
  if(prime)
  {
    printf("Prime!\n");
  }
  else
  {
    printf("Composite!\n");
  }
  mpz_clear(n);
  return (0);
}


//Pollard-Rho.cilk

#include <cilk-lib.cilkh>
#include <stdlib.h>
#include <stdio.h>
#include <gmp.h>
#include <time.h>
#include <miller-rabin.cilkh>

#define GRANULARITY 2

/*
 * Auxiliary function to do the heavy lifting. Sets "factor" to be the factor
it finds.
 * Returns 1 for success, 0 for failure.
 */
cilk int rho_factor_aux(mpz_t factor, mpz_t n, int c)
{
  int factored=0;
  mpz_t x,y,d;
  mpz_init_set_ui(x,2);
  mpz_init_set_ui(y,2);
  mpz_init_set_ui(d,1);

  while(mpz_cmp_ui(d,1)==0){
    mpz_powm_ui(x,x,2,n);
    mpz_add_ui(x,x,c);

    mpz_powm_ui(y,y,2,n);
    mpz_add_ui(y,y,c);
    mpz_powm_ui(y,y,2,n);
    mpz_add_ui(y,y,c);

    mpz_sub(d,x,y);
    mpz_abs(d,d);
    mpz_gcd(d, d, n);
  }
  if (mpz_cmp(d,n)==0)
  {
    mpz_set_ui(factor, 0);
  }
  else
  {
    mpz_set(factor, d);
    factored = 1;
  }
  mpz_clear(x);
  mpz_clear(y);
  mpz_clear(d);
  return (factored);
}

/*
```

```
 * Parallel for loop with granularity
 */
cilk int cilk_for_rho(mpz_t factor, int lo, int hi, mpz_t n)
{

  int mid, i, up, down, factored=0;
  mpz_t down_factor, up_factor;
  mpz_init(down_factor);
  mpz_init(up_factor);
  if(hi-lo>GRANULARITY)
  {
    mid = (lo+hi)/2;
    down = spawn cilk_for_rho(down_factor, lo, mid, n);
    up = spawn cilk_for_rho(up_factor, mid, hi, n);
    sync;
    if (down)
    {
      factored = 1;
      mpz_set(factor,down_factor);
    }
    else if (up)
    {
      factored = 1;
      mpz_set(factor, up_factor);
    }
    goto quit;
  }
  for (i = lo; i<hi; i++)
  {
    factored = spawn rho_factor_aux(factor, n,i);
    sync;
    if(factored)
    {
      abort;
      goto quit;
    }
  }

  quit:
    mpz_clear(down_factor);
    mpz_clear(up_factor);
    return (factored);
}

/*
 * Function that actually gets called
 * In this implementation, it just prints the factors as it finds them
 *   and repeats until all prime factors are found
 */
cilk int rho_factor(mpz_t n, int k)
{

  int factored, prime = 0;
  mpz_t factor, cofactor;
  mpz_init(factor);
  mpz_init(cofactor);
  while(1);
  {
    factored = spawn cilk_for_rho(factor, 1, k, n);
    sync;
    mpz_tdiv_q(cofactor, n, factor);
    prime = spawn miller_rabin(factor, k);
    sync;
    if (prime)
    {
      gmp_printf("Factor: %Zd\n", factor);
      mpz_tdiv_q(n, n, factor);
    }
    else
    {
      prime = spawn miller_rabin(cofactor, k);
      if (prime)
      {
        mpz_set(n, cofactor);
      }
  }
  return(1);
}

cilk int main(int argc, char *argv[])
{
  int prime, factor, k=10;
  mpz_t n;
  mpz_init_set_ui(n, 4725);
  if (argc > 2)
  {
    mpz_set_str(n, argv[1], 10);
    //k = atoi(argv[2]);
  }
  prime = spawn miller_rabin(n,k);
  sync;
  if(prime)
  {
    gmp_printf("%Zd is Prime!\n", n);
  }
  else
  {
    factor = spawn rho_factor(n, k);
    sync;
    //printf("Factor is: ", factor);
  }
  mpz_clear(n);
  return (0);
}
```