Implementing and Comparing Integer Factorization Algorithms

Jacqueline Speiser
jspeiser

Abstract

Integer factorization is an important problem in modern cryptography as it is the basis of RSA encryption. I have implemented two integer factorization algorithms: Pollard's rho algorithm and Dixon's factorization method. While the results are not revolutionary, they illustrate the software design difficulties inherent to integer factorization. The code for this project is available at https://github.com/jspeiser/factoring.

1 Introduction

The integer factorization problem is defined as follows: given a composite number N, find two integers x and y such that $x \cdot y = N$. Factoring is an important problem because if it can be done efficiently, then it can be shown that RSA encryption is insecure. For this project I have implemented two factoring algorithms: Pollard's rho algorithm and Dixon's factorization method. (I also implemented the quadratic sieve algorithm, but that code is not yet working.)

Pollard's rho algorithm [2] is a special-purpose factorization algorithm effective at factoring numbers with a small prime factor. It works by generating a sequence $x_1 = g(2), x_2 = g(g(2)), x_3 = g(g(g(2))), \ldots$ for some function g such as $g(x) = (x^2+1) \bmod N$. The algorithm runs two of these sequences at once, with one running "twice as fast" as the other. Eventually the sequences will reach a cycle and collide, at which point we can take the gcd of the two points and potentially retrieve a non-trivial factor. If no factor is found, we can restart the algorithm with either a new function g or a new seed.

Dixon's factorization method [1] is a general-purpose integer factorization algorithm. It works as follows: First, choose a bound B (the optimal runtime is achieved by choosing $B = \exp(\sqrt{\log N \log\log N})$) and let the *factor base* be the set of all primes smaller than B. Next, search for positive integers x such that $x^2 \bmod N$ is B-smooth, meaning that all the factors of x^2 are in the factor base. For all B-smooth numbers $x_i^2 = p^{e_1} p^{e_2} \ldots p^{e_k}$, record $(x_i^2, \vec{e}_i)$. After we have enough of these relations, we can solve a system of linear equations to find some subset of the relations such that $\sum \vec{e}_i = \vec{0} \bmod 2$. (See the Implementation section for details on how this is done.) Note that if k is the size of our factor base, then we only need $k+1$ relations to guarantee that such a solution exists. We have now found a congruence of squares, $a^2 = x_i^2$ and $b^2 = p_1^{\sum_i e_i 1} \ldots p_k^{\sum_i e_i k}$. This implies that $(a+b)(a-b) = 0 \bmod N$, which means that there is a 50% chance that $gcd(a-b, N)$ factors N. The runtime of Dixon's factorization method is $\exp(\sqrt{2 \log N \log\log N})$.

The quadratic sieve [3] is an optimization of the stage of Dixon's factorization method that searches for B-smooth numbers. Dixon's generates B-smooth numbers by randomly sampling numbers and checking whether they can be factored by the factor base. This is a very slow operation since we have to perform at least B arithmetic operations for every number being checked. The quadratic sieve uses the fact that $f(x)$ being B-smooth is equivalent to $f(x) = 0 \mod p_i$ for $f(x) = x^2 - N$ and small primes p_i to "sieve" for smooth numbers in a manner similar to the Sieve of Eratosthenes. The quadratic sieve improves the runtime of Dixon's to $\exp(\sqrt{\log N \log\log N})$.

2 Implementation

The code for this project along with instructions for how to build and run it are available at https://github.com/jspeiser/factoring. The files dixons.cc and pollard.cc both contain a `factor` function that could be used as library functions, but for ease of demonstration they also have `main` functions that can be invoked from the command-line.

Note that these implementations are pure implementations of the algorithms in question; they do not contain any logic for factoring numbers with multiple prime factors or for sanity checks such as whether N is even. This was a purposeful decision so that users of the library have the flexibility to heuristically decide which algorithms to use for numbers of various sizes.

Since Pollard's rho algorithm is so simple, this section will focus on implementation details of Dixon's factorization method. The only interesting thing to note about my implementation of Pollard's rho algorithm is that the user can specify a maximum number of seeds to try factoring N with before returning that there are no factors.

2.1 Gaussian Elimination

The Gaussian elimination step of Dixon's algorithms was a classic example of something that is mathematically simple but non-obvious how to implement in practice. Recall that during Dixon's factorization method we must solve a binary matrix to find a linear combination of equations such that their sum equals zero. The solving of a binary matrix is itself very straightforward: keep a matrix of bits and transform it into an upper triangular matrix by swapping and XORing rows as necessary. In my implementation, each row is an array of `uint64_t`s as large as necessary to represent the factor base (each bit of a `uint64_t` represents a column of the matrix). The trouble comes in when trying to construct the set of B-smooth values whose combination results in the zero row left at the bottom of the matrix. Since rows are swapped during Gaussian elimination and we are interested in combinations of rows instead of the actual solution to the system of linear equations, there is no clear way to reconstruct the information that we need.

To deal with this issue, I chose to keep a mapping of x_i's to x_j's with the number of times x_j had been added to x_i during the Gaussian elimination process. At the end of the algorithm, any x_j with an odd count is considered to be part of the linear combination that resulted in the final x_i row.

As an optimization, the Gaussian elimination function `findLinearDependencies` returns all of the linear combinations that result in $\vec{0}$. This was an attempt to reduce the number of times that the algorithm has to iterate before finding a nontrivial factor. Since generating B-smooth numbers is the bottleneck in Dixon's and Gaussian elimination becomes faster with each solved row, I believe that this optimization makes more sense than terminating Gaussian elimination as soon as there is a row of zeros. This would be an interesting area of experimentation, especially with the quadratic sieve optimization.

2.2 Parallelization

There are two main opportunities for parallelization in Dixon's factorization method: the generation of B-smooth numbers and Gaussian elimination. I parallelized both.

Parallelizing the generation of B-smooth numbers is simple in Dixon's factorization method since candidates are generated at random. I simply create the maximum number of threads specified by the user (the default is 8) and have those threads generate B-smooth numbers until I have as many as desired.

Parallelizing Gaussian elimination is a little trickier. The obvious intuition is to parallelize the processing of each row, but since every row can potentially affect every other row this does not work. Instead, I parallelize only the adding of rows. In Gaussian elimination, when processing a row i we look for the first unprocessed row whose ith column is a 1 and swap it with the target row. We then XOR this row with any row with a 1 in the ith column, thus ensuring that column i is 0 after row i. This process of addition can be parallelized, since at this point the rows are independent. I divide up rows $i+1$ to the bottom of the matrix evenly among the available threads, and only proceed to the next target row when all additions are complete.

2.3 Big Integers

The code available on the master branch of the github repository does all operations with `uint64_t`s. Obviously, this is too small for anything close to the scale of RSA moduli. For that reason, there is a bigint branch on the repository that supports factoring integers of arbitrary size. This branch uses the `boost::multiprecision` library to perform any operations with the potential to overflow a `uint64_t`. The branch is functional, but slow to the point of being unusable. Future development on this project will have to look into more efficient ways of dealing with arbitrarily large integers than the `boost` library provides. (Alternatively, the speed degradation could be mitigated by parallelizing across multiple machines instead of a single one.)

While not desirable in the long term, constraining the code to `uint64_t`s provided an interesting set of challenges regarding overflow. Supporting the factoring of integers up to the maximum that can fit in a `uint64_t`

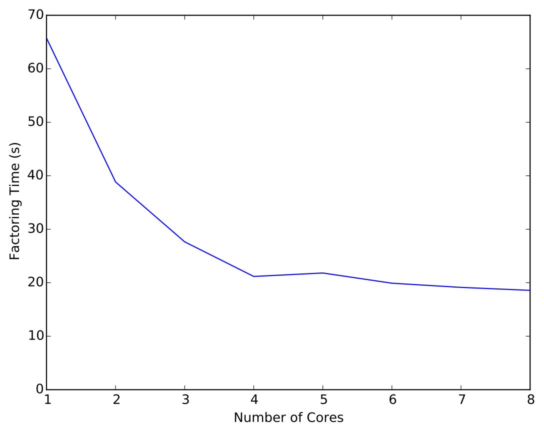

Figure 1: Time to factor 773,978,585,664,881 using Dixon's factorization method with different numbers of cores.

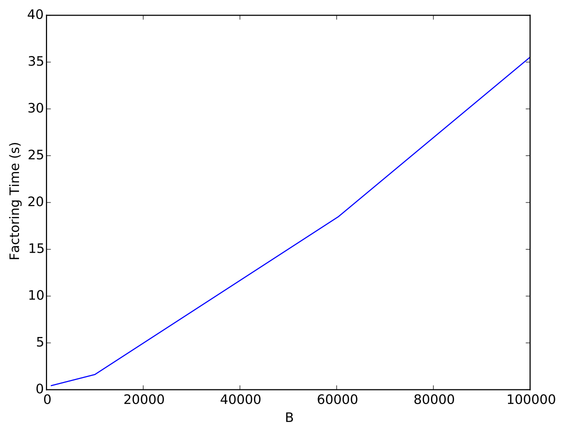

Figure 2: Time to factor 773,978,585,664,881 using Dixon's factorization method with different choices of B.

means that almost any intermediate operation during the factoring process is likely to overflow. To deal with this issue I used special `mulmod` and `modpow` functions designed to avoid overflow.

3 Results

Because using arbitrary precision integers slows down my implementation on Dixon's to the point of being unusable, the numbers that I present here are for the implementation that uses `uint64_ts`.

3.1 Comparison

Factoring $982,451,629 \cdot 982,451,653 = 965,211,226,903,592,737$ with Dixon's takes 8 minutes, 3.64 seconds (on 8 cores with $B = 247,108$ and 1 iteration). Factoring the same number with Pollard's takes 0.036 seconds. This discrepancy is consistent with the fact that Pollard's is supposed to be more efficient on small prime factors.

3.2 Parallelization in Dixon's

Figure 1 shows the time to factor $15,485,863 \cdot 49,979,687 = 773,978,585,664,881$ with various numbers of cores (this number was chosen because it is large enough to see clear performance differences but small enough that testing was relativly quick). My machine has 4 physical cores and 8 if you include hypertwins, which explains why the performance improvement levels off with more than 4 cores. I suspect that the improvement between 1 and 4 cores is not quite linear due to issues related to thread creation overhead and cache coherence/locality. Instead of maintaining dedicated worker threads, the code currently spins up new threads every time there is new work to complete. Additionally, there are a couple of data structures used in Gaussian elimination that would probably benefit from resturcturing to ensure that different threads operate on different cache lines.

3.3 Choice of B in Dixon's

Figure 2 shows the time to factor 773,978,585,664,881 with different values of B (all on 8 cores). The theoretically optimal value of B for this number is 60356, but the graph shows that performance is actually much better with smaller values of B. I suspect that this choice of N is simply too small for the chance of finding a trivial factor to be an issue.

4 Conclusion and Future Work

Working on this project brought up several interesting issues regarding software design of theoretical algorithms. First, machine architecture has a significant impact on performance. Arbitrary precision libraries slow down basic arithmetic operations significantly, making it infeasible to factor large numbers on a typical laptop. To scale to larger integers would require either a cluster of many machines working in parallel or an architecture with a larger word size (128- or 256-bits). Second, theoretical guidelines on parameter values are not necessarily what should be used in practice. These should be determined experimentally, as they likely depend on architecture and implementation details. Finally, there are performance bottlenecks that theory does not account for, such as cache coherency.

There is a lot of room for future work on this project. The obvious next step is to finish implementing the quadratic sieve and compare it to Dixon's. After that, I think it would be beneficial to figure out what is slow in `boost`'s arbitrary precision library, find another library, and/or implement such a library myself. Further comparison of factoring algorithms is not possible when constrained to the size of a `uint64_t`, since these algorithms are meant to be used on much larger numbers (the quadratic sieve is most efficient on numbers between 10^{50} and 10^{100}, for example). Finally, it would be interesting to investigate how clusters of machines can be used to parallelize further than multiple cores on a single machine.

References

[1] DIXON, J. D. Asymptotically fast factorization of integers. *Mathematics of computation 36*, 153 (1981), 255–260.

[2] POLLARD, J. M. Theorems on factorization and primality testing. In *Mathematical Proceedings of the Cambridge Philosophical Society* (1974), vol. 76, Cambridge Univ Press, pp. 521–528.

[3] POMERANCE, C. A tale of two sieves. *Biscuits of Number Theory 85* (2008), 175.