CADO-NFS, a Number Field Sieve implementation

S. Bai[0], P. Gaudry[1], A. Kruppa[1], F. Morain[2], L. Muller[1], E. Thomé[1], **P. Zimmermann**[1]

[0] AUSTRALIAN NATIONAL UNIVERSITY ;
[1] CARAMEL/INRIA/LORIA ; [2] TANC/INRIA/LIX

Dec 18th, 2011

(credit E. Thomé for most slides)

Motivations

Integer factorization ($N = pq \longrightarrow$ find p, q) is a hard problem.

- Pre-1980's: a stumbling block in mathematical computations, and a challenging problem. Some significant advances in the 1970's.
- 1978-present: IF has attracted considerable attention because of its relevance for cryptography through the RSA cryptosystem.

CADO-NFS: an implementation of NFS

The fastest integer factoring algorithm is the Number Field Sieve.

- Very complicated algorithm. Embarks lots of number theory. (much more involved than, e.g., the ECM factoring algorithm)
- Very few available implementations. State of the art is at best bits and pieces from here and there.

CADO project. Write our own code. Joint effort, started in 2007.

- Actively developed. Playground for new ideas.
- Certainly beatable, but contains nice algorithms.
- No refrain to reorganizing the code to (changing) taste every so often.

CADO-NFS is LGPL, and written (almost) entirely in `C`. To date, ~ 120 kLOC.

Objectives for an NFS program

An NFS program like CADO-NFS can be used for various purposes.

- "below-NFS-threshold" numbers. Below 100dd, QS is faster.
  $\Rightarrow$ intended for routine checking, timings are not the issue.
- Numbers which explore the limitations of the current code.
  Do growing sizes, add optimizations. Ongoing effort.
- Record-size numbers. CADO-NFS can't factor `rsa768` yet, but participating to `rsa768` taught us a lot.

Note: CADO-NFS is clearly not an integrated factoring machinery.
CADO-NFS does not include ECM, QS, ...

- No interaction with a user.
- Interface: a collection of programs driven by a main script.

Record sizes: crypto in sight

The feasibility limit explored by NFS records is used to determine key sizes for RSA.

- SSL/TLS. In 2011, 512-bit certificates are still in use (about 2%)!!! See `http://ssl.entrust.net/blog/?p=1041`.

- EMV credit cards (a.k.a. chip and pin).

  Most chip public keys are 960b. Some 1024b (until end of 2009, some had a 896b key).

Factoring experiments: decision-driving data for setting key sizes.

The polynomial selection phase

For factoring "general" N, GNFS uses:

- an irreducible polynomial $f \in \mathbb{Z}[x]$;
- another irreducible polynomial g in $\mathbb{Z}[x]$ such that f and g have a common root modulo N

Usually f has degree 5 or 6, and g is linear.

$g = x - m$: f corresponds to the base-m decomposition of N

$g = px - m$: f corresponds to the base-m/p decomposition of N

g defines the rational side, f defines the algebraic side.

General plan: Obtain relations, and combine them to obtain:

$$x^2 \equiv y^2 \mod N.$$

Recognizing when $a - b$ factors

Essentially, we want that the integers

$$F(a,b) := b^d f(a/b) \qquad \text{and} \qquad G(a,b) = b^1 g(a/b)$$

are simultaneously smooth.

Complexity of NFS

For factoring an integer N, GNFS takes time:

$$L_N[1/3, (64/9)^{1/3}] = \exp\left((1 + o(1))(64/9)^{1/3}(\log N)^{1/3}(\log\log N)^{2/3}\right).$$

This is sub-exponential.

Note: some special numbers allow for a faster variant NFS, with complexity

$$L_N[1/3, (32/9)^{1/3}] = \exp\left((1 + o(1))(32/9)^{1/3}(\log N)^{1/3}(\log\log N)^{2/3}\right).$$

NFS: no panic

NFS might not be the simplest algorithm on earth, but:

- obstructions have been dealt with already long ago. See literature.
- the bottom line is simple: everything boils down to assembly/C/MPI.

Polynomial selection: find f, g;

Sieving: find many a, b s.t. $F(a, b) = b^d f(a/b)$ and $G(a, b)$ smooth.

Linear algebra: combine a, b pairs to get a congruence of squares.
($\Rightarrow$ solve a large sparse linear system over $\mathbb{F}_2$.)

Square root: complete the factorization.

Recent progresses

Since RSA-155 (512 bits) in 1999, many improvements.

- Much better polynomial selection (Kleinjung, 2003, 2006).
- Very efficient sieving code (Franke, Kleinjung, 2003–).
- Very efficient cofactorization code (Kleinjung, Kruppa).

More recent state of the art, notably for linear algebra:

- Use block Wiedemann algorithm (BW), at separate locations.
- Use computer grids idle time to do linear algebra.
- Use sequences of unbalanced length in BW.

Polynomial selection

Asymptotic analysis of NFS gives formulae for:

- asymptotic optimal value for $\deg f$ (for an n-bit number).
- asymptotic optimal value for the coefficient sizes.

Trivial "base-m" approach:

- Choose the degree d. Choose an integer $m \approx N^{1/(d+1)}$;
- Write N in base m: $N = f_d m^d + f_{d-1} m^{d-1} + \cdots + f_0$.
- Pick $f = f_d x^d + \cdots + f_0$ and $g = x - m$.

We have an immense freedom in the choice of $m \Rightarrow$ can do better.

Polynomial selection algorithms

Algorithms aim at polynomial pairs (f, g) s.t. $F(a, b) = b^d f(a/b)$:

- is comparatively small over the sieving range.
- is often smooth (f with many roots mod small p).

Several relevant algorithms:

- Kleinjung (2006): handle an immense amount of possible polynomials, explore promising ones.
- Murphy (1999): rotation and root sieve: $(f, g) \rightsquigarrow (f + \lambda g, g)$.
- Kleinjung (2008): modification of the 2006 algorithm.

CADO-NFS has a `polyselect` program implementing this.

- polynomial root finding mod small p;
- knapsack-like problem solving;
- sieving for good λ; could use GPUs.

Sieving: a very old tool

In order to find (a, b) pairs for which $F(a, b)$ is smooth:

- For all small primes p (or prime powers) ;

Sieving: a very old tool

In order to find (a, b) pairs for which $F(a, b)$ is smooth:

- For all small primes p (or prime powers) ;
- for all roots r of f mod p, pick (a_0, b_0) s.t. $a_0 \equiv rb_0 \mod p$;

Sieving: a very old tool

In order to find (a, b) pairs for which $F(a, b)$ is smooth:

- For all small primes p (or prime powers) ;
- for all roots r of f mod p, pick (a_0, b_0) s.t. $a_0 \equiv rb_0 \mod p$;
- for all (u, v), mark $(a_0 + pu, b_0 + pv)$ as being divisible by p.

Keep (a, b) pairs which have been marked most.

Do this on both sides (f and g). Deciding in which order in subtle.

Note: NFS computation time is mostly spent on sieving.

Typical problems with sieving

There are several practical shortcomings.

- The (a, b) space to be explored is large, but predicting in advance the yield for a range of (a, b) pairs is hard ;
- The yield drops as (a, b) grow ;
- $\Rightarrow$ diminishing returns.

Lattice sieving to the rescue.
Old idea (1993), but superiority demonstrated only after 2000.

Lattice sieving: how do we sieve ?

Given a prime q and $\begin{pmatrix} e_0' \\ e_1' \end{pmatrix} = \begin{pmatrix} a_0 & b_0 \\ a_1 & b_1 \end{pmatrix}$, we consider the lattice

$$L_q = \{(a, b) = ie_0' + je_1'\}.$$

All work is done on the (i, j) plane. A rectangle $R_{(i,j)}$ is fixed. The workplan for sieving for this *special* q is:

- Describe locations to sieve in the (i, j) plane.
- Sieve "small" factor base primes.
- Sieve "large" factor base primes.
- Do this for both sides.
- Locations which have been marked most need to be factored.

Fine points of sieving

For a given q, explore some $R_{(i,j)}$ of size e.g. 2^{31}.

- Divide into areas matching L1 cache size (64kb typically), to be processed one by one.
- Small primes hit often: once per row.
- Larger primes hit rarely. Rather maintain a "schedule" list to circumvent cache misses: "bucket sieving".
- Use multithreading.

CADO-NFS implements this in `las`.

- Hot spots in assembly; Use vector instructions when relevant;
- Optimize some data structures to reduce memory footprint;
- Strive to eliminate badly predictable branches;
- POSIX threads;
- Factoring good (a, b)'s: Use $p \pm 1$ and special-purpose ECM.

Fast forward

The output of the sieve process is a set of relations.

These undergo:

- Filtering: making a small relation set from a large one ;
- After filtering, linear system solving.

Algorithmically, nothing very new in filtering since Cavallar (2000).

Implementation in CADO-NFS:

- Hash tables all over the place;
- Minimum spanning trees to help decision;
- Has supported MPI distribution at some point;

Does the job so far.

Linear algebra

Must combine relations so that they consist of only squares.

This rewrites as a linear system. (everything reduces to lin. alg. !)

- matrix M: a relation appears in each row. Coefficients are multiplicities of prime factors (and ideals). Most are zero.
- A vector v such that

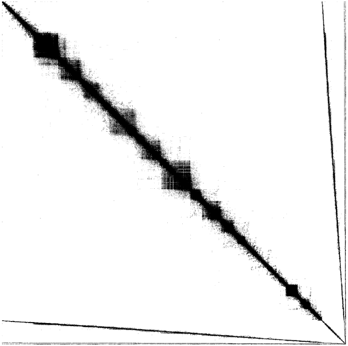
  $$vM = 0 \mod 2$$

  indicates which relations to combine in order to obtain only squares (even multiplicities).

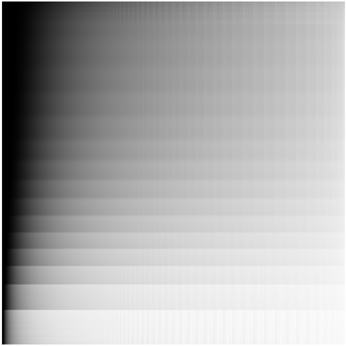
Equivalently, we rephrase this as a linear system $Mv = 0$ (transposing M).

Note: linear algebra mod 2 differs much from linear algebra over $\mathbb{C}$.

$\mathbb{F}_2$ is exact, and positive characteristic



(some PDE example)



(a factoring matrix)

Linear algebra

We have an $N \times N$ matrix M. We want to solve $Mv = 0$.

The matrix M is large, (very) sparse, and defined over $\mathbb{F}_2$.

Because of sparsity, we want a black box algorithm.

There are several sparse linear algebra algorithms suitable for $\mathbb{F}_2$:

- Lanczos ;
- Wiedemann ; others.

These early suggestions are unsuitable. Bit arithmetic: slow. Also, failure probability $1/\#\mathbb{F}_2 = 1/2$ is not so tempting...

Block algorithms

Block algorithms apply the black box to e.g. $n = 64$ vectors at a time. (n is prescribed by the hardware)

- Block Lanczos (BL). $\frac{2N}{n-0.76}$ black box applications ;
- Block Wiedemann (BW). $\frac{3N}{nn'}$, n' times (n' small).

BL is appealing if one has a large cluster.

BW is preferred since it offers distribution opportunities.

Block Wiedemann: workplan

- Initial setup. Choose starting blocks of vectors x and y.
- Sequence computation. Want L first terms of the sequence:

$$a_i = x^T M^k y.$$

  - Computing one term after another, this boils down to our black box $v \mapsto Mv$.
  - This computation can be split into several independent parts (which all know M).
- Compute some sort of minimal polynomial.
- Build solution as:

$$v = \sum_{k=0}^{\deg f} M^k y f_k.$$

  - Again, this uses the black box.
  - Can be split into many independent parts (which all know M).

Linear algebra: size matters

The matrix M itself is soon out of reach for core storage.

- 2005: kilobit SNFS: 64M rows/cols, 10G non-zero coeffs. About 30GB.
- 2010: 768b GNFS: 192M rows/cols, 27G non-zero coeffs. About 75GB.

Computing Mv is also a lot of work. Try to use many processors if possible.

This is a classical HPC concern.

- Split the matrix into equal parts.
- Exploit high-bandwith channels: shared memory, infiniband network.

Features of the CADO-NFS BW code

CADO-NFS has a complete BW implementation.

Sequence computation:

- POSIX threads;
- MPI – implementation agnostic. Some optimized collectives;
- Some kind of "sparse binary BLAS" used. Assembly;
- (Stem of) capability to switch to other base field;
- Mostly `C`, some `C++`. Wrapper script in Perl.

Minimal polynomial computation using a quasi-linear algorithm.

- recursive structure;
- arithmetic on matrices of polynomials over $\mathbb{F}_2$.
- very old code, needs rework.

The square root step

Our congruence of squares actually comes as:

$$(a_1 - b_1 m)(a_k - b_k m)\phi((a_1 - b_1)(a_k - b_k)) \mod N.$$

- Both sides are known to factor with even multiplicities: they are squares.
- BUT computing the square root is in fact non trivial (esp. on algebraic side).

CADO-NFS implements quasi-linear algorithms for this

- Newton lifting.
- Arithmetic modulo fixed degree polynomials.
- Suitable for current records.
- Alternative algorithm (waives a number theoretic assumption):
  - Explicit CRT.
  - Can be distributed with MPI.

There exists a more advanced square root algorithm for this step (Montgomery), but it needs more software support.

Breakdown of code size

Utility libraries	35 kLOC (not all is used)
Polyselect	17 kLOC
Sieving	20 kLOC (some dead)
Filtering	8 kLOC
Linear algebra	33 kLOC (some generated)
Square root	9 kLOC

Conclusion and further work

Many points would be interesting to improve.

- Polyselect with GPUs (but `msieve` does this already).
- Lattice siever needs cleanup, and some obvious improvements.
- Filtering currently can't handle record sizes.
- Filtering should use several threads.
- Linear algebra sparse BLAS can be improved.
- Linear algebra minimal polynomial step must be reworked.
- The whole chain could be adapted to discrete log computation.