Parallelizing the F4 Algorithm

Roman Pearce

axcas.net

Abstract. Computing a Gröbner basis can be the first step in solving a system of polynomial equations exactly. We describe a high performance multithreaded implementation of the F4 algorithm of Faugère for primes up to 63-bits. Our implementation in C is available at `axcas.net` and is released into the public domain, with the goal of seeing it incorporated into computer algebra systems, both open source and commercial.

1 Introduction

A Gröbner basis for a set of polynomials F under a monomial ordering $<$ is a set of polynomials $G = \{g_1, \ldots, g_t\}$ such that division under $<$ by G (in any order) produces unique remainders. They are a useful tool for solving many problems in algebraic geometry but they are expensive to compute. Most computer algebra systems have an implementation of *Buchberger's algorithm* [1], but in 1999, J.C. Faugère [2] described the F_4 algorithm which is much more efficient.

Faugère's original algorithm, like Buchberger's, spends a significant amount of time reducing polynomials to zero. Faugère followed up by proposing F_5, which uses *signatures* to predict and avoid reductions to zero [3], but in order to do it the Gröbner basis must be computed incrementally. That is, a basis for $\{f_1, f_2\}$ must be computed in full before f_3 is added. For some problems the intermediate bases are larger than the final basis, in particular, for systems with no solutions where the final basis is $\{1\}$.

A different approach was taken by Allan Steel for Magma and by Monagan and Pearce for Maple. They used *Monte-Carlo Gaussian elimination* to largely eliminate reductions to zero in F_4 [8]. This made F_4 competitive with F_5, and it seems to have been widely adopted. Further progress has been made through parallelization and vectorization, leading to high performance implementations that can compute Gröbner bases for cyclic-10 modulo p in a matter of minutes. We present one such implementation here, with benchmarks in Section 3.

2 Computing Gröbner Bases

We show how Gröbner bases are computed using a small example. Let $f(x, y) = x^2 + y$ and $g(x, y) = xy - 1$. We order terms first by their total degree, with ties broken by degree in x. Starting with the set $F = \{f, g\}$, a *syzygy* is found by canceling the least common multiple of x^2 and xy which is x^2y.

$$\begin{array}{rl} y \cdot f(x,y) = & +x^2y + y^2 \\ -x \cdot g(x,y) = & -x^2y \qquad + x \\ \hline h(x,y) = & \qquad\quad +y^2 + x \end{array}$$

In general the terms of a syzygy may be reducible by $f(x,y)$ or $g(x,y)$ and we would want to take its remainder. The result is either zero or a new polynomial, in this case $h(x,y)$, with a leading term that is not reducible. We add this to our basis and consider two new syzygies with h:

$$\begin{array}{rl} y^2 \cdot f(x,y) = & +x^2y^2 \qquad + y^3 \\ -x^2 \cdot h(x,y) = & -x^2y^2 - x^3 \\ \hline Syz(f,h) = & \qquad -x^3 + y^3 \\ +x \cdot f(x,y) = & \qquad +x^3 \qquad + xy \\ -y \cdot h(x,y) = & \qquad\qquad -y^3 - xy \\ \hline = & \qquad\qquad\qquad 0 \end{array}$$

$$\begin{array}{rl} y \cdot g(x,y) = & +xy^2 \qquad - y \\ -x \cdot h(x,y) = & -xy^2 - x^2 \\ \hline Syz(g,h) = & \qquad -x^2 - y \\ +f(x,y) = & \qquad +x^2 + y \\ \hline = & \qquad\qquad 0 \end{array}$$

The set $G = \{f, g, h\}$ is a Gröbner basis because all syzygies reduce to zero. This is Buchberger's algorithm, which reduces syzygies one at a time and adds non-zero remainders to the basis, generatating additional syzygies as it goes.

The F_4 algorithm improves on this by processing several syzygies at once. It builds a matrix whose columns correspond to the monomials that appear in the syzygies and divisions. We show the matrix below for $\{f, g, h\}$ and the syzygies between f and g (first two rows) and g and h (second two rows). Below the syzygies come the reductors: f reduces x^2 and g reduces y^2.

$$\begin{array}{r|rrrrrr} & x^2y & xy^2 & x^2 & y^2 & x & y \\ \hline y \cdot f & 1 & 0 & 0 & 1 & 0 & 0 \\ x \cdot g & 1 & 0 & 0 & 0 & -1 & 0 \\ y \cdot g & 0 & 1 & 0 & 0 & 0 & -1 \\ x \cdot h & 0 & 1 & 1 & 0 & 0 & 0 \\ f & 0 & 0 & 1 & 0 & 0 & 1 \\ h & 0 & 0 & 0 & 1 & 1 & 0 \end{array}$$

Because the columns of the matrix are sorted in the term ordering, we expect Gaussian elimination to mimic the process of polynomial division. In the matrix above the syzgies all reduce to zero because $\{f, g, h\}$ is a Gröbner basis, however in general we will find rows with new leading monomials which are added to the basis, generating new syzygies.

Computing a Gröbner basis with rational numbers suffers from *intermediate expression swell* [5], so our implementation is modulo a machine prime $p < 2^{63}$. We expect computer algebra systems to use Chinese remaindering and rational reconstruction [11].

In the 63-bit case, multiplication mod p uses Möller and Granlund's 2 by 1 division with a precomputed reciprocal [6]. This has performance comparable to Montgomery multiplication [9].

2.1 F_4 Overview

Below is an outline of F_4 that we refer to as we describe our implementation.

```
Algorithm F4
Input : an array of polynomials F = [f1,...,fs], a monomial ordering <
Output: a Groebner basis G for the ideal <f1,...,fs> with respect to <
  G := [];  # the partial basis
  P := [];  # the array of syzygy pairs
  M := [];  # the array of monomials in A
  for each f in F do
    G, P := update_pairs(f);           # form initial basis and pairs
  while |P| > 0 do
    A, P := select_pairs(P);           # get syzygies of least degree
    A, M := symbolic_preprocessing(A); # build description of matrix
       A := encode_matrix(A, M);       # encode sparsity pattern of A
       A := row_reduction(A);          # sparse Gaussian elimination
       A := new_pivot_rows(A);         # select rows with new pivots
       A := back_substitution(A);      # inter-reduce new pivot rows
       B := convert_to_polys(A, M);    # convert rows to polynomials
    for each b in B do
       G, P := update_pairs(b);        # generate new syzygy pairs
  return inter_reduce(G);              # produce a reduced basis
```

The data structure `f4row` is used to represent polynomials and matrix rows. It contains a length, a monomial multiplier, a pointer to an array of monomials, a pointer to an array of coefficients modulo p, and a pointer to encoded column indices. The terms are sorted in descending order.

Monomials are 32-bit integers referring to a position inside a block of memory. For each monomial we store an exponent vector and column, using 32-bit words. Monomials are hashed in a global table to ensure uniqueness.

The syzygy pairs are stored in a structure `f4syz` containing pointers to two rows and the lcm of their leading monomials. At the beginning of each iteration, we select pairs whose lcm is smallest in the ordering [2].

Symbolic preprocessing loops over the rows and multiplies each monomial by the multiplier. When the product is a new monomial, it searches the basis for a divisor and if one is found it creates a new row.

Encoding the matrix assigns column indices to the monomials and creates a compressed list of indices for each row. The encoding stores the initial index as a machine word, followed by non-zero differences as bytes provided they are 255 or less. A zero byte indicates the end of a run, and a new initial index follows unless we are at the end of a row. This encoding was first used by Faugère [2].

Fig. 1. A sequence of column indices is encoded.

$$\begin{array}{|c|c|c|c|c|c|}\hline 10000 & 9990 & 9989 & 5000 & 4950 & \dots \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|c|c|c|c|}\hline 10000 & 10 & 1 & 0 & 5000 & 50 & \dots \\ \hline \end{array}$$

Encoding the matrix adds to the cost of F_4 in two ways. First, monomials are multiplied twice, once in symbolic preprocessing and again while encoding. Second, row reduction must decode rows to perform the linear algebra.

Row reduction has an array of pointers to pivot rows and a buffer. The first step is to assign rows to be pivots, preferring the sparsest rows. The remaining rows must be reduced using the pivots. To reduce a row we decode its contents into the buffer, loop down the entries, and subtract multiples of pivot rows. Any non zero result becomes a new pivot row.

The probabilistic algorithm divides the N rows to be reduced into blocks of size $3\sqrt[3]{N}$ and reduces random linear combinations of each block, stopping after a zero reduction occurs. Compared to blocks of size $\sqrt{N}$, this approach favours manycore processors but does a bit more work in total.

We dynamically limit the number of pairs to control the size of the matrices. When selecting pairs of degree d, we initially limit the number of pairs to 2048. The pair limit doubles in the next step if the algorithm is still processing pairs of degree d. In this way we capture both early degree drops and make good use of probabilistic elimination when there are a very large number of pairs. We use Gebauer and Möller's strategy to eliminate pairs [4].

2.2 Parallelization

Measurements of the sequential time provide an estimate of the gains available through parallelization or vectorization. In the table below, cyclic-9 spends two thirds of its time in row reduction. If we speed up row reduction by a factor of eight the overall speedup is expected to be $1/(.33 + .66/8) = 2.42$ or less due to overhead. We parallelized symbolic preprocessing, encoding, row reduction, and back substitution based on data like this.

Table 1. The percentage of time spent in various parts of F_4.

$p = 2^{63} - 25$	select/gc	symbolic	encode	reduce	backsub	decode	update
cyclic-9	1.20%	14.1%	9.9%	66.30%	4.80%	0.1%	3.53%
gametwo	0.53%	9.5%	7.4%	68.94%	10.05%	0.1%	3.40%
jason210	3.06%	63.4%	22.9%	7.20%	2.53%	0.02%	0.89%
katsura-12	0.23%	13.8%	13.3%	69.18%	2.52%	0.5%	0.94%
mayr42	8.57%	21.6%	2.1%	0.34%	0.13%	0.001%	66.84%
noon-9	0.68%	47.7%	17.6%	17.39%	2.29%	0.05%	14.08%

One may ask why we did not parallelize the Gebauer and Möller strategy as it clearly matters on some problems. The answer is that this is difficult, and we hope to address the problem in future work.

Our approach to parallel symbolic preprocessing is new. In an outer loop we examine the remaining rows to be processed and preallocate space for new rows and new monomials. Threads are spawned to process those rows, with new rows added to the matrix via lock free *compare and swap*. On dense problems most of the monomial products are already present in the hash table, so the creation of any new monomials is guarded by a lock. When the lock is acquired the table must be searched again.

Row reduction is often the most expensive part of F_4 and many authors have parallelized it. Our code follows [7] by adding rows to the pivot array using compare and swap. Each thread acquires a block of rows to reduce by atomically incrementing a counter using compare and swap. It proceeds to reduce random linear combinations of the rows, and when a non-zero remainder is obtained, it tries to add the remainder to the pivots array using compare and swap. If this fails the row is reduced further before trying again.

3 Benchmarks

We used a Ryzen 9955HX with 16 cores, 32 threads, and 96GB of DDR5 RAM running Debian Linux 13. We compared axf4 to msolve 0.9.4, GamBa 0.2, and Singular 4.4.1. For msolve we used the option -l 42 which gave the fastest time.

Table 2. Benchmarks of Gröbner bases modulo machine primes.

$p = 2^{31} - 1$	axf4 st	axf4 32t	msolve 1t	msolve 32t	gamba	singular
cyclic-9	42.956s	7.120s	37.690s	26.100s	12.600s	3737.014s
katsura-12	18.347s	2.215s	13.800s	6.410s	5.610s	2726.490s
noon-9	10.688s	6.760s	5.260s	8.160s	3.860s	14.846s
$p = 2^{63} - 25$	axf4 st	axf4 32t	msolve 1t	msolve 32t	gamba	singular
cyclic-9	75.240s	9.897s	–	–	–	–
katsura-12	32.645s	3.399s	–	–	–	–
noon-9	11.475s	6.918s	–	–	–	–

For axf4, the parallelization is efficient but the single threaded throughput is not as good as we would like. To improve the times substantially we would have to support a vector instruction set, with little potential gain for 63-bit primes. Adding vectorization to our highly portable program is a task for future work. Presumably, GamBa is is gaining almost a factor of four from AVX2.

We wanted to measure the efficiency of the programs on a larger problem so we disabled SMT and used `/usr/bin/time -v` for measurements.

Table 3. Benchmark of cyclic-10 with hyperthreading disabled.

$p = 2^{31} - 1$	axf4 st	axf4 16t	msolve 1t	msolve 16t	gamba
user time	4388.92	7047.51	4333.80	12374.02	613.93
wall time	1:13:15	9:50.73	1:12:18	24:17.01	10:17.85
percent cpu	99%	1195%	99%	849%	99%
max resident kb	26589080	36371332	25353084	25588196	20273128
minor page faults	6675659	9380452	5224677	5105278	2244560

Next, we take a look at our historical ability to compute Gröbner bases over three decades on a midrange PC. The 1994 machine is lost to time, but testing suggests it should have been able to compute cyclic-8. The machines are:
1994 Pentium 90 Mhz, 16MB RAM, 32-bit DOS extender
2004 iBook G4 1.07 GHz, 768MB RAM, 32-bit Mac OS X 10.5
2014 Pentium G3250 (Haswell) 3.2 GHz, 16GB RAM, 64-bit Windows 10
2024 Core i5 1340p (Raptor Lake) 4.6 GHz, 64GB RAM, 64-bit Debian Linux

Table 4. axf4 single threaded performance to 2024, plus 16 threads and gamba.

$p = 2^{31} - 1$	1994	2004	2014	2024	2024 16t	2024 gamba
cyclic-8	maybe	75.690s	3.385s	1.157s	0.623s	0.530s
cyclic-9	–	4788.580s	146.388s	57.378s	14.629s	22.700s
cyclic-10	–	–	15144.471s	5573.007s	1749.212s	1178.610s
cyclic-11	–	–	–	–	*no	*maybe

* allowed up to 900GB of NVMe Gen 4 swap (7000MB/s)

Our performance on 2024 machine is better than it looks. The Core i5 1340p has 12 cores: 4 hyperthreaded p-cores which turbo up to 4.6GHz and 8 e-cores which turbo up to 3.4GHz. It is not clear how much parallel speedup to expect. Worse, under full load the cores run at 2.7GHz and 2.5GHz respectively, which explains our poor showing on cyclic-10.

We can see that the performance of single threaded C code is hitting a wall. On the 2014 machine, going from 32-bit to 64-bit code produces a speedup of a factor of two. Vector instructions offer a similar gain for the 2024 machine, but the real problem is memory. The amount of RAM available increased by a factor of 48, then 21, then 4. At first glance we are going to need new algorithms to go on computing Gröbner bases, but fast NVMe storage offers some relief.

We wanted to compute cyclic-11 on the 2024 machine but the computation uses too much memory, so we reconfigured the Ryzen with 2TB of NVMe Gen5 swap (14900MB/s). The faster drive, extra RAM, and more cores makes a big difference on large steps with no degree drop such as the one shown below.

```
STEP 85
degree=18 pairs=65536/172489 with pairs limit 65536
2354716 x 2681314 with 60329639623 non-zero, 25620.8 per row
1.020 bytes per non-zero, matrix encoded in 58659.404 MB
131072 rows to reduce, blocksize 150, max threads 874, zero reductions 1
new=6520 basis=94711, step time=105892.505 sec
```

Our final time on cyclic-11 mod $p = 2^{31} - 1$ was
seconds, and the basis
had
polynomials and was
GB.

4 Conclusion

We thank Guillem Fernandez, the author of GamBa, for productive discussions. In the future we hope to vectorize our software and increase CPU utilization on machines with a large number of cores. Intel has a chip with 288 vectorized e-cores and AMD has a chip with 192 high performance cores, so we expect the major limitation over the next decade will be RAM. We were able to overcome this bottleneck on our PC by using NVMe storage for swap.

References

1. Cox, Little, O'Shea, Ideals, Varieties, and Algorithms. Springer, 1996.
2. J.C. Faugère, A new efficient algorithm for computing Gröbner bases (F4), Journal of Pure and Applied Algebra, Volume 139, Issues 1–3, pp. 61–88, 1999.
3. J.C. Faugère, A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In Proc. ISSAC 2002, ACM Press. pp. 75–83, 2002.
4. R. Gebauer, M. Möller, On an installation of Buchberger's algorithm, Journal of Symbolic Computation, Volume 6, Issues 2–3, Pages 275-286, 1988.
5. Geddes, Czapor, Labahn, Algorithms for Computer Algebra. Kluwer, 1992.
6. N. Möller and T. Granlund, Improved Division by Invariant Integers, in IEEE Transactions on Computers, vol. 60, no. 2, pp. 165–175, Feb. 2011.
7. M. Monagan, R. Pearce, A Compact Parallel Implementation of F4. Proc. PASCO 2015, pp. 95–100, 2015.
8. M. Monagan, R. Pearce, An Algorithm For Spliting Polynomial Systems Based On F4. Proc. PASCO 2017, Article 12, Pages 1–5, 2017.
9. P. Montgomery, Modular Multiplication Without Trial Division. Mathematics of Computation. 44 (170): 519–521, April 1985.
10. J. Berthomieu, C. Eder, M. Safey El Din. msolve: A Library for Solving Polynomial Systems. Proc. ISSAC 2021, pp. 51–58, 2021.
11. P. Wang. A p-adic Algorithm for Univariate Partial Fractions. Proc. SYMSAC '81, ACM Press, pp. 212–217, 1981.