

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 $G = \{g_1, \dots, g_t\}$ 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* [2], but in 1999, J.C. Faugère [3] 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 [4], 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 [11]. 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 .

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

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 :

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

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, generating 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 reducers: f reduces x^2 and g reduces y^2 .

	x^2y	xy^2	x^2	y^2	x	y
$y \cdot f$	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

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 syzygies 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* [8], so our implementation is modulo a machine prime $p < 2^{63}$. We expect computer algebra systems to use Chinese remaindering and rational reconstruction [13] to compute bases over $\mathbb{Q}$.

In the 63-bit case, multiplication mod p uses Möller and Granlund's 2 by 1 division with a precomputed reciprocal [9]. This has performance comparable to Montgomery multiplication [12].

2.1 F_4 Overview

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

Algorithm F_4

```

Input : an array of polynomials  $F = [f_1, \dots, f_s]$ , a monomial ordering  $<$ 
Output: a Groebner basis  $G$  for the ideal  $\langle f_1, \dots, f_s \rangle$  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
     $G, P := \text{update\_pairs}(f)$ ; # form initial basis and pairs
while  $|P| > 0$  do
     $A, P := \text{select\_pairs}(P)$ ; # get syzygies of least degree
     $A, M := \text{symbolic\_preprocessing}(A)$ ; # build description of matrix
     $A := \text{encode\_matrix}(A, M)$ ; # encode sparsity pattern of  $A$ 
     $A := \text{row\_reduction}(A)$ ; # sparse Gaussian elimination
     $A := \text{new\_pivot\_rows}(A)$ ; # select rows with new pivots
     $A := \text{back\_substitution}(A)$ ; # inter-reduce new pivot rows
     $B := \text{convert\_to\_polys}(A, M)$ ; # convert rows to polynomials
    for each  $b$  in  $B$  do
         $G, P := \text{update\_pairs}(b)$ ; # generate new syzygy pairs
return  $\text{inter\_reduce}(G)$ ; # 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.

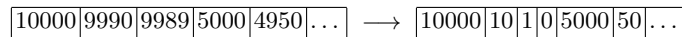
Monomials are 32-bit integers referring to a position inside a block of memory. For each monomial we store an exponent vector and column, using 32-bit words. Monomials are hashed in a global table to ensure uniqueness.

The syzygy pairs are stored in a structure **f4syzy** 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 [3].

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 [3].

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



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 [7].

2.2 Parallelization

Measurements of the sequential time provide an estimate of the gains available through parallelization or vectorization. In the table below, cyclic9 spends 62% of its time in row reduction. If we speed up row reduction by a factor of eight the overall speedup is expected to be at most $1/((.38 + .62/8) = 2.18$. Based on data like this we parallelized symbolic preprocessing, encoding, row reduction, and back substitution.

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

$p = 2^{31} - 1$	gc	select	symbolic	encode	reduce	backsub	decode	update
bayes148	0.8%	0.4%	58.9%	27.5%	4.5%	3.3%	0.0%	4.5%
cyclic9	0.2%	0.3%	15.4%	15.4%	62.0%	3.3%	0.1%	3.1%
jason210	1.0%	1.8%	53.6%	26.4%	11.2%	3.3%	0.1%	0.8%
katsura13	0.0%	0.8%	25.4%	25.9%	43.7%	0.9%	0.1%	2.2%
mayr42	0.9%	2.2%	15.5%	1.6%	0.8%	0.2%	0.0%	78.5%
noon9	0.2%	0.9%	31.9%	29.7%	19.0%	4.3%	0.1%	13.3%
reimer8	0.0%	0.5%	37.3%	39.5%	21.1%	0.7%	0.0%	0.3%

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 a second time to prevent the creation of duplicate monomials.

Row reduction is often the most expensive part of F_4 and many authors have parallelized it. Our code follows [10] 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.

It should be noted that in our parallel routines, threads acquire work from a global structure as opposed to being assigned work in advance. This is key to obtaining good timings on small problems and small steps.

3 Benchmarks

We used a Ryzen 9955HX 16 core/32 thread machine with 96GB of DDR5 and Debian Linux 13 to compare axf4ver2g to msolve 0.9.5 and GamBa 0.3. Msolve used the option -l 44 which gave the fastest time on cyclic9.

Table 2. Benchmarks of Gröbner bases modulo machine sized primes.

	$p = 2^{63} - 25$		$p = 2^{31} - 1$					
	axf4 st	axf4 32t	axf4 st	axf4 32t	msolve 1t	msolve 32t	gamba avx2	gamba avx512
bayes148	38.316s	21.076s	38.355s	21.119s	11.27s	21.90s	4.73s	4.76s
cyclic9	74.801s	8.150s	42.187s	5.233s	31.70s	26.48s	12.32s	10.31s
jason210	4.172s	5.423s	4.091s	5.425s	1.75s	5.31s	1.19s	1.43s
katsura13	44.515s	4.468s	28.712s	3.096s	11.58s	10.70s	5.57s	5.77s
mayr42	68.365s	57.707s	68.416s	57.067s	21.46s	47.97s	10.36s	9.44s
noon9	8.518s	4.700s	7.911s	4.602s	3.51s	9.18s	3.92s	3.51s
reimer8	18.682s	2.247s	15.521s	1.834s	9.63s	9.36s	4.73s	5.44s

The timings show that the cost of 63-bit primes in axf4 is reasonable. For problems where linear algebra dominates (cyclic9, katsura13, see Table 1), axf4 has good parallel speedup. Our parallel timings generally go down except for on jason210 which is problematic. Nevertheless, axf4 may be slow at updating pairs (mayr42) or computing monomial operations generally. It should be noted that GamBa uses 16-bit exponents and may vectorize some monomial operations.

3.1 Cyclic-10 Benchmark

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. See Table 3.

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. One option is to use vectorization to support 128-bit or larger primes.

Table 3. Benchmark of cyclic10 with hyperthreading disabled.

$p = 2^{31} - 1$	axf4 st	axf4 16t	msolve 1t	msolve 16t	gamba avx2	gamba avx512
user time	4255.22	6734.34	3474.20	13296.40	595.52	503.08
wall time	1:10:59	8:37.43	57:58.15	24:12.21	9:59.13	8:26.81
percent cpu	99%	1303%	99%	915%	99%	99%
max resident kb	13894720	23880860	25368376	25321492	20320568	20595260
minor faults	3432148	6204609	5229652	6437458	2248200	2196828

3.2 Axf4 On Older Machines

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 cyclic8. 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 (avx2).

$p = 2^{31} - 1$	2004	2014	2024	2024 axf4 16t	2024 gamba
cyclic8	39.282s	2.968s	1.085s	0.495s	0.55s
cyclic9	2872.067s	124.491s	56.797s	11.731s	22.78s
cyclic10	—	10678.782s	5217.471s	1367.497s	1140.19s

Our performance on 2024 machine should be better. 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 relatively poor speedup on cyclic10 (3.8x versus 4.85x on cyclic9).

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.

3.3 Cyclic-11 Benchmark

We wanted to compute cyclic11 on the 2024 machine but the computation uses too much memory, so we reconfigured the Ryzen with 3TB of NVMe Gen5 swap (14900MB/s). The faster drive, extra RAM, and more cores made a big difference on large steps with no degree drop such as the one shown below.

```
STEP 135
degree=19 pairs=16440/184574 with pairs limit 131072
3741472 x 4106190 with 79248365332 non-zero, 21181.1 per row
1.031 bytes per non-zero, matrix encoded in 77956.993 MB
32880 rows to reduce, blocksize 96, max threads 343, zero reductions 1
new=1701 basis=102417, step time=49905.165 sec
```

It should be pointed out that a Gröbner basis for cyclic11 was first computed by Faugère in [6] using an algorithm that divides out by an Abelian group, and by Allan Steel in 2016 using a dense F4 variant modulo 32003.

Our time on cyclic11 mod $p = 2^{31} - 1$ was 519201.445 seconds and the computation used 1.1TB of memory. The basis has 30320 polynomials and is about 15 GB. The step above shows a limitation of our approach, because the encoded matrix threatens to spill out of RAM. If that were to occur, the overhead of swapping might become too great.

4 Conclusion and Future Work

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 to some extent by using NVMe storage for swap.

In the meantime we have implemented FGLM [5] and the Gröbner Walk [1] for converting Gröbner bases to lexicographical order, however these programs are not parallelized. We plan to release and report on them in the near future.

References

1. S. Collart, M. Kalkbrener, and D. Mall. Converting bases with the gröbner walk. *Journal of Symbolic Computation*, 24(3-4):465–469, 1997.
2. D. Cox, J. Little, and D. O’shea. *Ideals, varieties, and algorithms*. Springer, 1997.
3. J.-C. Faugere. A new efficient algorithm for computing gröbner bases (f4). *Journal of pure and applied algebra*, 139(1-3):61–88, 1999.
4. J. C. Faugere. A new efficient algorithm for computing gröbner bases without reduction to zero (f5). In *Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, pages 75–83, 2002.

5. J.-C. Faugere, P. Gianni, D. Lazard, and T. Mora. Efficient computation of zero-dimensional gröbner bases by change of ordering. *Journal of Symbolic Computation*, 16(4):329–344, 1993.
6. J.-C. Faugère and J. Svartz. Gröbner bases of ideals invariant under a commutative group: the non-modular case. In *Proceedings of the 38th international symposium on symbolic and algebraic computation*, pages 347–354, 2013.
7. R. Gebauer and H. M. Möller. On an installation of buchberger’s algorithm. *Journal of Symbolic computation*, 6(2-3):275–286, 1988.
8. K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for computer algebra*. Springer, 1992.
9. N. Möller and T. Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60(2):165–175, 2010.
10. M. Monagan and R. Pearce. A compact parallel implementation of f4. In *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation*, pages 95–100, 2015.
11. M. Monagan and R. Pearce. An algorithm for splitting polynomial systems based on f4. In *Proceedings of the international workshop on parallel symbolic computation*, pages 1–5, 2017.
12. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.
13. P. S. Wang. A p-adic algorithm for univariate partial fractions. In *Proceedings of the fourth ACM symposium on Symbolic and algebraic computation*, pages 212–217, 1981.