Parallelizing the F4 Algorithm for Gröbner Bases

Roman Pearce

July 29, 2025

Abstract

Computing a Gröbner basis can be the first step in solving a system of polynomial equations exactly. In this article we describe the F4 algorithm of Faugère along with its optimizations, and show how it can be parallelized effectively for multi-core computers. Our implementation in C is available at `axcas.net` and is released into the public domain.

1 Using the Software

On the website `axcas.net` under "Code" you will find a link to the F4 Algorithm. Download the file `axf4.zip` and extract its contents into a directory. The software is distributed as source code which must be compiled with a C compiler. Detailed instructions are given in the file `readme.txt`, however once you have installed a C compiler you should be able to open a development terminal and run `make` on Windows, Mac, or Linux. The compiler generates an executable program and you can run one of the examples as follows:

```
axf4 -p 9223372036854775783 -v [x0,x1,x2,x3,x4,x5,x6] cyclic7.txt
```

This line specifies a machine-sized prime, $p = 2^{63} - 25$, an ordered list of variables, and a file containing the input polynomials with one polynomial per line. The program will run F4 and generate an output file `cyclic7.txt.out` containing a reduced Gröbner basis for the graded reverse lexicographical ordering.

2 Gröbner Bases

Given a set of polynomials $F = \{f_1, f_2, \ldots, f_s\}$, a Gröbner basis $G = \{g_1, g_2, \ldots, g_t\}$ with respect to a *monomial ordering* $<$ has the following nice properties:

- Division by G produces unique remainders.
- The solutions and their multiplicities are the same for G as for F.
- The number of monomials not reducible by G is the same number of solutions.
- If $<$ orders first by degree, one can compute the dimension of the set of solutions from G.
- Algorithms such as FGLM [7] or the Gröbner Walk [3] can be run on G to eliminate variables or triangularize the system in preparation for factorization or solving.

For details and proofs we refer you to [4]. For an exposition of FGLM and the Gröbner Walk, see [5]. For additional algorithms such as primary decomposition, see [1].

By way of a small example, we will show how a Gröbner basis can be computed. Let $f(x, y) = x^2 + y$ and $g(x, y) = xy - 1$. We will 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.

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

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 by any leading term of F. We add this to our basis and consider two new syzygies involving h and their respective remainders.

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

The set $G = \{f, g, h\}$ is then a Gröbner basis because all syzygies reduce to zero. We have in fact used *Buchberger's algorithm* [4] to compute the Gröbner basis. This algorithm reduces syzygies one at a time and adds non-zero remainders to the basis, generatating additional syzygies as it goes.

3 F4 Algorithm

The F4 algorithm [6] improves on Buchberger's algorithm by processing several syzygies at a time. It builds a matrix whose columns correspond to the monomials that appear in the syzygies and divisions. We show the matrix below for $G = \{x^2 + y, xy - 1, y^2 + x\}$ and the syzygies between G_1 and G_2 (first two rows) and G_2 and G_3 (second two rows). Below the syzygies come the reductors: G_1 reduces x^2 and G_3 reduces y^2.

	x^2y	xy^2	x^2	y^2	x	y
$y \cdot G_1$	1	0	0	1	0	0
$x \cdot G_2$	1	0	0	0	-1	0
$y \cdot G_2$	0	1	0	0	0	-1
$x \cdot G_3$	0	1	1	0	0	0
G_1	0	0	1	0	0	1
G_3	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 syzgies all reduce to zero because G is a Gröbner basis, but in general we will find rows with new leading monomials after Gaussian elimination. These polynomials are then added to the basis and new syzygies are generated. Below is an outline of F4 that we will 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);              # select syzygies of least degree
    A, M := symbolic_preprocessing(A);    # build description of the matrix
       A := encode_matrix(A, M);          # encode the 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
```

4 Implementation

Like the division algorithm and the Buchberger's algorithm, the F4 algorithm over the rationals suffers from intermediate expression swell. Because of this we have implemented F4 over the integers modulo a prime where $p < 2^{63}$, with the idea that computations over the rationals should use the Chinese remainder theorem and rational reconstruction [9, 11].

The data structure `f4row` is used to represent both 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 always sorted in descending order.

Monomials are stored as integers referring to a position inside a contiguous block of memory. Inside the block of memory, we store for each monomial an exponent vector and column index. Monomials are hashed in a global hash table to ensure they are unique.

The syzygy pairs are stored in a structure `f4syz` containing pointers to two rows and the least common multiple of their leading monomials. At the beginning of each iteration, we select syzygy pairs whose lcm is smallest in the ordering, following the *normal* strategy of Buchberger [2] and Faugère [6].

4.1 Symbolic Preprocessing

Symbolic preprocessing loops over all matrix rows, multiplying each monomial by the monomial multiplier for that row. When the product is a new monomial not previously seen, it searches the basis for a divisor and if one is found it creates another new row. We use the column index stored after the exponent vector to mark monomials as processed.

```
Symbolic Preprocessing
Input  : an array A of F4 rows
         an array B of basis elements
Output : an array M of monomials, updates A
  M := [];
  for i from 0 to |A|-1 do
    for j from 0 to A[i].length do
      m := multiply(A[i].multiplier, A[i].monomial[j]);
      if Column(m) is set then continue;
      set Column(m) := 1; and add m to M;
      for k from 0 to |B|-1 do
        if B[k].monomial[0] divides m then
          create a new F4 row with
            length      := B[k].length;
            multiplier  := m/(B[k].monomial[0]);
            monomial    := B[k].monomial;      # copies pointer
            coefficient := B[k].coefficient;   # copies pointer
          add the new row to the end of A;
          break;
  return A, M;
```

4.2 Encoding the Matrix

The first step of encoding the matrix is to sort the array of monomials from symbolic preprocessing and assign column indices to each monomial. Then for each row of the matrix, we store a list of column indices for row reduction. 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. An example is shown below. This encoding was first used by Faugère [6].

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

Encoding the matrix adds to the cost of F4 in two ways. First, all of the monomials need to be multiplied twice, once in symbolic preprocessing and again during encoding once the column indices are known. Second, during Gaussian elimination, the rows must be decoded to perform linear algebra.

4.3 Gaussian Elimination

Row reduction is implemented using two arrays, each as long as the number of columns in the matrix. One array stores pointers to the pivot rows and the other is a buffer that is used for row reduction. Our first step is to assign as many rows as possible to be pivots, preferring the sparsest rows. The remaining rows must be reduced using the pivots. In general, the pivot rows are created by symbolic preprocessing and the rows to be reduced come from the syzygies. So if there are k pairs selected, there should be $N = 2k$ rows to be reduced with respect to a large matrix that is upper triangular.

To reduce a row, we decode its contents into the buffer, loop down the entries, and subtract pivot rows when they exist. When the result is not zero we obtain a new pivot. In Gröbner basis computations most of these rows reduce to zero.

We employ a trick first used by Allan Steel in Magma and Monagan and Pearce in Maple [10] which makes the F4 algorithm Monte Carlo. We divide the N rows to be reduced into blocks of size $\sqrt{N}$, and reduce random linear combinations of each block. The probability of obtaining k zero reductions by chance is about $1/p^k$, so when this is acceptable we discard the block. Our implementation uses a bound of 10^{-18}, which means that if $p = 2^{31} - 1$ two zero reductions are required to discard a block, and if $p = 2^{61} - 1$ then only one zero reduction is required. This makes the algorithm an order of magnitude faster in practice. If the prime and block size are too small, i.e. if $1/p^{\sqrt{N}} > 10^{-18}$, then this technique is not used.

4.4 Pair Limits

An important modification to F4 is to limit the number of pairs, and therefore the size of the matrices in each step. For homogeneous computations the degree of the pairs increases monotonically, and rebuilding matrices slows down the algorithm. But for many practical problems, e.g. cyclic-n roots, there are steps where the degree drops and this technique can save time. The idea was suggested to us by Allan Steel.

We implemented a dynamic pair limit as follows. 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 Monte Carlo Gaussian elimination when there are a very large number of pairs.

4.5 Updating Pairs

Buchberger provides two criteria that substantially reduce the number of pairs that need to be considered. First, if the lcm of the leading monomials is equal to their product, the pair reduces to zero. Second, if the lcm of the leading monomials of `B[i]` and `B[j]` is reducible by some `B[k]`, with $\{i, j, k\}$ distinct, and both pairs `(B[i],B[k])` and `(B[j],B[k])` are not present, then the pair `(B[i],B[j])` can be discarded [4].

We prefer to use Gebauer and Möller's strategy [8]. When inserting a new polynomial `B[k]` into the basis, we generate pairs `(B[i],B[k])` for $i < k$ where the leading monomials are not relatively prime. From earlier pairs `(B[i],B[j])` we delete any whose lcm is reducible by `B[k]` but not equal to the lcm for `B[i]` and `B[k]` or `B[j]` and `B[k]`. Among the new pairs `(B[i],B[k])` we delete any whose lcm is divisible by, but not equal to, the lcm for `(B[j],B[k])` for $j \neq i$. Finally, among the new pairs `(B[i],B[k])` we delete any whose lcm is exactly equal to the lcm for `(B[j],B[k])` with $j < i$.

5 Parallelization

In parallel programming, programs start threads which can run on different CPU cores at different rates, and co-ordinate using atomic operations that affect shared memory. *Compare and swap* is the primary tool we use, because it updates values atomically and acts as a memory barrier, which stops the compiler and CPU from reordering operations across it.

```
Compare and Swap (atomic)
Input  : a pointer P, an old value A, a new value B.
Output : the value stored at P originally
  T := *P;               # T is the value pointed to by P
  if (T = A) *P := B;    # update memory if this value is A
  return T;              # return the original value
```

Suppose we start threads hoping to process the rows of a matrix in parallel. Each thread could run a function like the one below, which acquires rows atomically and stops when all rows have been acquired.

```
Worker Thread
Input : a pointer B to bounds in shared memory (start and end)
      st := B[0]; sz = B[1];
get:  so := compare_and_swap(&B[0],st,st+1);
      if (so != st) then
        st := so;   # update start position and try again
        goto get;
      else if (so >= sz) then
        return ;    # no more work left to do so return
      ... work on matrix row st ...
      goto get;     # get another row
```

5.1 Parallel Gaussian Elimination

Reducing rows is often the most expensive part of F4 and has been parallelized in Maple, Magma, and libraries such as GBLA. Our code follows the Maple approach of adding rows to the pivot array using compare and swap [10]. Each thread acquires a block of rows using the technique of the previous section. It constructs random linear combinations of those rows, reducing each combination using the pivots.

When a non-zero remainder is obtained, the thread uses compare and swap to assign it to the pivots array. On failure, the row is copied back into the buffer and reduced further before trying again.

5.2 Parallel Symbolic Preprocessing

Symbolic preprocessing is difficult to parallelize because it creates monomials and adjusts the size of the matrix in the innermost loop. Our approach is to run the algorithm in stages, where the number of new monomials and matrix rows are bounded in each stage by the free space in the monomial hash table. The hash table is kept under 50% full for performance reasons.

```
Symbolic Preprocessing in Parallel
Shared : a hash table H of monomials
         an array A of F4 rows
         an array B of basis elements
         an array M of monomials
Output : an array M of monomials, updates A
  M := []; i := 0;
  while i < |A| do
    L := 0; j := i;           # count terms left in A
    while j < |A| and L < H.free do
       L := L + A[j].length;
    H := enlarge(H,L);        # add space for L new monomials
    A := enlarge(A,L);        # add space for L new rows
    bounds := [i, j];
    N := number_of_cpus();
    for k from 1 to N do tid[k] := start_thread(symbolic_worker, &bounds);
    for k from 1 to N do join_thread(tid[k]);
    i := j;                   # |A| is updated by the threads
  return A, M;
```

The symbolic worker function is shown below. It acquires a new monomial by setting the column index with compare and swap, and then it proceeds to divide by the partial basis. Where we have written to do something atomically, it means using the technique introduced at the beginning of Section 5.

```
Symbolic Preprocessing Worker
Shared : an array A of F4 rows
         an array B of basis elements
         an array M of monomials
  acquire the next row A[i] atomically
  for j from 0 to A[i].length do
    m := multiply(A[i].multiplier, A[i].monomial[j]);
    if compare_and_swap(&Column(m),0,1) then continue;
    add m to M atomically;
    for k from 0 to |B|-1 do
      if B[k].monomial[0] divides m then
        create a new F4 row with
          length      := B[k].length;
          multiplier  := m / (B[k].monomial[0]);
          monomial    := B[k].monomial;       # copies pointer
          coefficient := B[k].coefficient;    # copies pointer
        add the new row to the end of A atomically;
        break;
  return;
```

Our final problem is that monomials must be created in parallel. We allow the hash table to be searched in parallel, but the creation of a new monomial is done in a critical section. When the lock is acquired the table is searched again to ensure correctness. The hash table is Davenport's power of two quadratic hash.

```
Create Monomial in Parallel
Input  : N the number of variables
         S the size of the hash table which is a power of two
         an array E containing the exponents for the new monomial
Shared : an array T where T[k] is the hash for the value T[S+k]
        h := hash(E);
retry:  k := h;
        for i from 0 to S-1 do
          k := (k+i) mod S-1;
          if (T[k] = 0) break;
          if (T[k] != h) continue;
          m := T[S+k];
          V := exponent_vector(m);
          for i from 0 to N-1 do
            if (V[i] != E[i]) break;
          if (i = N) return m;
        if (compare_and_swap(&table_lock,0,1)) goto retry;
        for i from 0 to S-1 do      # search table again
          k := (k+i) mod S-1;
          if (T[k] = 0) break;
          if (T[k] != h) continue;
          m := T[S+k];
          V := exponent_vector(m);
          for i from 0 to N-1 do
            if (V[i] != E[i]) break;
          if (i = N) goto done;
        m := monomial_count++;
        V := exponent_vector(m);
        for j from 0 to N-1 do  V[j] := E[j];
        Column(m) := 0; T[s+k] := m;
        compare_and_swap(&T[k],0,h);
done:   table_lock := 0;
        return m;
```

6 Benchmarks

Benchmarking software is becoming more difficult these days with variable clock speeds, heterogeneous cores, and simultaneous multithreading. Our first benchmark was run with 32-bit code on a Raspberry PI 4B with 8 GB of RAM, which has none of those features. We used `/usr/bin/time -v` for measurements.

cyclic-9 mod $p = 2^{31} - 1$	sequential	1 thread	2 threads	4 threads
real seconds	595.69	594.11	327.13	199.23
cpu seconds	593.49	591.32	617.49	681.05
cpu utilization	99%	99%	189%	343%
max memory (KB)	176060	181220	214048	238180
parallel speedup	–	1.003x	1.821x	2.990x

This benchmark shows that our parallel code nearly matches the efficiency of the sequential code when one thread is used. For two threads the result is also good, but for four threads we can see the code can not fully utilize the cpu and there is a substantial increase in cpu time as well. Part of the inefficiency is due to Amdahl's Law, where some steps in the algorithm are too short to parallelize effectively. But it looks like four threads are exhausting the resources of the Broadcom BCM2711 CPU. This CPU has 1 MB of shared L2 cache for the cores to communicate, and that bottleneck might account for the increase in CPU time.

Our remaining benchmarks use a more powerful computer, a Ryzen 9955HX with 16 cores, simultaneous multithreading which we disabled in the Linux kernel, and 64 MB of shared L3 cache. The maximum clock speed is 5.4 GHz. The computer has 96GB of DDR5 (5600) RAM and a Gen5 7000MB/s NVMe swap drive.

We selected three problems to stress different apsects of the F4 algorithm: cyclic-9 is a generic problem with degree drops, katsura-12 emphasizes linear algebra, and noon-9 emphasizes symbolic preprocessing. We ran problems with a 31-bit prime $2^{31} - 1$ and a 63-bit prime $2^{63} - 25$. The software is compiled for 64-bit in both cases, but uses more efficient linear algebra code for smaller primes.

cyclic-9 31-bit p	sequential	1 thread	2 threads	4 threads	8 threads	16 threads
real seconds	53.61	60.04	32.80	20.29	12.83	8.75
cpu seconds	53.47	59.16	59.08	62.52	63.88	71.26
cpu utilization	99%	99%	180%	309%	503%	831%
max memory (KB)	294996	323092	378364	457904	560176	757740
parallel speedup	–	0.893x	1.634x	2.642x	4.178x	6.127x

cyclic-9 63-bit p	sequential	1 thread	2 threads	4 threads	8 threads	16 threads
real seconds	82.00	89.62	47.72	28.04	16.87	11.11
cpu seconds	81.87	88.75	88.80	92.54	93.71	104.33
cpu utilization	99%	99%	186%	331%	559%	951%
max memory (KB)	294720	323988	374512	455096	525656	590416
parallel speedup	–	0.915x	1.718x	2.924x	4.861x	7.381x

katsura-12 31-bit p	sequential	1 thread	2 threads	4 threads	8 threads	16 threads
real seconds	27.62	27.34	16.11	9.70	6.49	4.21
cpu seconds	27.49	26.92	29.39	30.73	32.13	35.32
cpu utilization	99%	99%	184%	321%	515%	883%
max memory (KB)	177452	225116	253456	276624	320304	383028
parallel speedup	–	1.010x	1.714x	2.847x	4.256x	6.561x

katsura-12 63-bit p	sequential	1 thread	2 threads	4 threads	8 threads	16 threads
real seconds	36.50	38.49	21.46	12.32	7.70	5.07
cpu seconds	36.34	38.00	40.13	40.97	42.05	45.58
cpu utilization	99%	99%	188%	336%	558%	940%
max memory (KB)	176084	223356	256256	275544	299416	351156
parallel speedup	–	0.948x	1.701x	2.963x	4.740x	7.199x

noon-9 31-bit p	sequential	1 thread	2 threads	4 threads	8 threads	16 threads
real seconds	15.86	18.47	13.05	10.23	7.74	6.83
cpu seconds	15.76	18.14	19.80	22.08	25.68	36.91
cpu utilization	99%	99%	152%	217%	336%	547%
max memory (KB)	228212	238952	241432	254672	291360	355584
parallel speedup	–	0.859x	1.215x	1.550x	2.049x	2.322x

noon-9 63-bit p	sequential	1 thread	2 threads	4 threads	8 threads	16 threads
real seconds	15.90	18.57	13.51	10.44	7.59	6.85
cpu seconds	15.82	18.23	20.44	22.70	25.77	37.28
cpu utilization	99%	99%	152%	219%	343%	552%
max memory (KB)	229284	240852	241276	262148	300520	384504
parallel speedup	–	0.856x	1.177x	1.523x	2.095x	2.321x

7 Conclusion

We have presented a parallel implementation of F4 modulo a prime that has reasonable speedup on most problems which are dominated by linear algebra. We consider the parallel speedup obtained so far to be "pretty good" because the Gröbner basis computations consist of many steps, some of which are too small or quick to parallelize effectively.

As an aside, we were able to compute a Gröbner basis for cyclic-10 on the Ryzen 9955HX in under 12 minutes for $p = 2^{31} - 1$, but cyclic-11 is proving to be much harder. We hope to report on this in due time.

References

[1] Becker and Weispfenning. Gröbner Bases. Springer (1993).

[2] Bruno Buchberger. An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. J. Symbolic Comp., 41(3-4):475–511, (2006). Translated from the 1965 German original by Michael P. Abramson.

[3] S. Collart, M. Kalkbrenner, and D. Mall. Converting bases with the Gröbner walk. J. Symbolic Comp. 6, 209–217 (1997).

[4] Cox, Little, and O'Shea. Ideals, Varieties, and Algorithms. Springer (1996).

[5] Cox, Little, and O'Shea. Using Algebraic Geometry. Springer (2004).

[6] Jean-Charles Faugère. A New Efficient Algorithm for Computing Grobner Bases (F4). Journal of Pure and Applied Algebra, 139 (1). (1999)

[7] J.C. Faugère; P. Gianni; D. Lazard; T. Mora. Efficient Computation of Zero-dimensional Gröbner Bases by Change of Ordering. Journal of Symbolic Computation. 16 (4): 329–344. (1993)

[8] Rüdiger Gebauer, H. Michael Möller. On an Installation of Buchberger's Algorithm. Journal of Symbolic Computation, Vol 6 issue 2–3 (1988).

[9] M. Monagan. Maximal quotient rational reconstruction: An almost optimal algorithm for rational reconstruction. Proceedings of ISSAC 2004. 243–249, (2004).

[10] Michael Monagan, Roman Pearce. A Compact Parallel Implementation of F4. Proceedings of PASCO 2015.

[11] P. S. Wang. A p-adic Algorithm for Univariate Partial Fractions. Proceedings of SYMSAC '81, ACM Press, pp 212-217, (1981).