A Compact Parallel Implementation of F4

Michael Monagan
Department of Mathematics
Simon Fraser University
Burnaby, B.C. V5A 1S6, CANADA.
mmonagan@cecm.sfu.ca

Roman Pearce
Department of Mathematics
Simon Fraser University
Burnaby, B.C. V5A 1S6, CANADA.
rpearcea@cecm.sfu.ca

ABSTRACT

We present a compact and parallel C implementation of the F4 algorithm for computing Gröbner bases which uses Cilk. We give an easy way to parallelize the sparse linear algebra which is the main cost in practice. To obtain more speedup we attempted to parallelize the generation of sparse matrices as well. We present timings to assess the effectiveness of our approach and to compare our implementation to others.

1. INTRODUCTION

Gröbner bases are a useful tool in computer algebra that can also be expensive to compute. Thus, they are a natural target for parallelization. There are three main algorithms for computing Gröbner bases: the Buchberger algorithm [1], the F4 algorithm by Faugère [4], and the F5 algorithm also by Faugère [5]. The ideas in F5 have also been generalized to produce several other algorithms, see [3] for a survey.

Attempts to parallelize Gröbner basis computations have a long history; see Vidal [17], Revees [15], and Leykin [10]. Recently, authors have focused on parallelizing the sparse linear algebra used in F4 and F5. Faugere and Lachartre [6] partition the matrix into regions $[[A \mid B], [C \mid D]]$ where A is upper triangular, the columns of A and C correspond to the monomials that are reducible, and we compute $D - CA^{-1}B$ and triangularize it to obtain new basis elements. A similar scheme is proposed by Neumann in [13]. An advantage of our approach is that it is very easy to implement.

In fact, a goal of this paper is to show how to implement a version of F4 that is compact, fast, and easy to parallelize. Our C library is under 30KB and computes Gröbner bases modulo a 31-bit prime. Some additional Maple code allows us to compute over $\mathbb{Q}$ via Chinese remaindering and rational reconstruction. We obtain good performance and reasonable parallel speedup in both cases.

Our paper is organized as follows. Section 2 describes the F4 algorithm and our approach to implementing it. Section 3 presents benchmarks and an assessment of our approach.

2. F4 ALGORITHM

To compute a Gröbner basis, the F4 algorithm inserts all of the initial polynomials into the basis and generates a set of S-pairs. It then proceeds in five steps:

1. A subset of S-pairs are selected, normally those having lowest lcm degree. For each $S(g_i, g_j) = X \cdot g_i - Y \cdot g_j$ it constructs the polynomials $X \cdot g_i$ and $Y \cdot g_j$.
2. Symbolic preprocessing loops over all the polynomials and for each reducible monomial m, divisible by the leading term of some g_i, it constructs $(m/LM(g_i)) \cdot g_i$ and loops over that polynomial as well.
3. The set of monomials encountered in step 2 is sorted in the term order, the position becomes a column index, mapping the polynomials to rows of a matrix.
4. Gaussian elimination is performed on the matrix.
5. Rows whose leading monomial is not divisible by the basis are added to the basis; new pairs are generated.

To remove redundant pairs we use the Gebauer and Möller criteria [7].

Faugère described two optimizations in [4]. First, he uses a simplify function to rewrite polynomials. Given $X \cdot g_i$ he searches previous matrices for a row $Y \cdot g_i$ with $Y|X$. If one is found, he replaces $X \cdot g_i$ with $(X/Y) \cdot h$ where h is taken from the row reduced matrix and $LM(h) = LM(Y \cdot g_i)$. He then simplifies $(X/Y) \cdot h$ recursively if $X \neq Y$.

Second, Faugère adopts a complicated storage scheme to compress the matrices. He stores the coefficients of each g_i and reuses them for each $X \cdot g_i$ (changing only the indices), which saves a factor of two in space. He also suggests that rather than store the indices, one can store their differences which often fit into bytes. That saves up to a factor of four over 32-bit indices.

In an early version of our software we experimented with associating a polynomial to each monomial and simplifying as suggested by Faugère, but we did not observe a speedup. We decided against implementing Faugère's storage scheme because it would complicate the code and we do not appear to need the memory savings. Recall that Faugére computed Gröbner bases of cyclic-9 in the late 1990's and faced severe memory constraints. A typical machine of the era may have only 500 MB of RAM. Nowadays, we can "throw hardware at the problem" and use solid state hard drives for swap. In our opinion, large Gröbner basis computations are probably limited more by time than memory despite the introduction of multicore CPUs.

2.1 Polynomial Representation

Polynomials in $\mathbb{Z}_p[x_1, ..., x_n]$ with $p < 2^{31}$ are represented as arrays of 32-bit integers. The first word stores the length of the array, followed by pairs of monomials and coefficients sorted in a monomial ordering. Borrowing an idea from the Magma computer algebra system, monomials are an index into a block of memory, and hashing is used to make them unique [16]. Once a monomial is created it persists over the course of an entire Gröbner basis computation. We tried to garbage collect them, but this did not improve our timings.

Attached to each monomial we store its degree, which is used for comparisons, a hash value to prevent recalculation, a cache for divisions by the basis, and a column index that is used to map polynomials to matrix rows for elimination.

To hash monomials $x_1^{e_1} \cdots x_n^{e_n}$ we compute $h = \sum_{i=1}^{n} r_i e_i$ where the r_i are random integers chosen at the outset. The hash table size is a power of two, and we use the quadratic probing method of Hopgood and Davenport [8]. The value of h is stored in the monomial to accelerate probes.

We tested the following idea, due to Faugere and used in Magma [16]. When multiplying two monomials, their hashes are added and a lookup is performed to find the product in the table if it exists. The scheme is very effective at finding products and it made monomial multiplications 20% faster.

Our main goal in designing the monomial representation was to accelerate symbolic preprocessing, which consists of dividing each monomial by the Gröbner basis, and for those which divide, multiplying the basis element by a monomial. In each monomial we store an index into the basis to let us continue dividing where we left off. After row reduction, we use the index to identify rows whose leading element is not reducible by the basis. These rows are added to the basis.

Finally, the monomials contain a field for a column index. After symbolic preprocessing, we collect all the monomials that appear in the matrix, sort them into descending order, and assign them indices. The column index field is used to map monomials to column indices for Gaussian elimination.

2.2 Gaussian Elimination

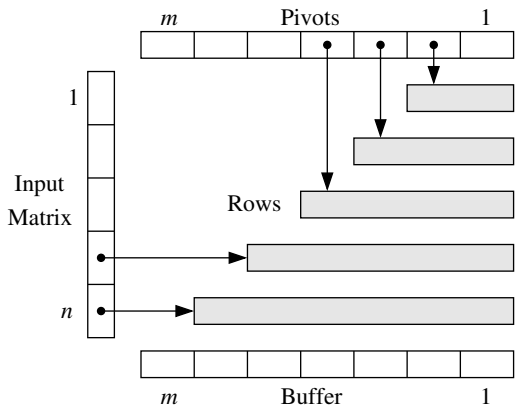
The main source of parallelism in F4 is from sparse linear algebra. This takes the majority of time, and as we will see, it is relatively simple to parallelize. An $n \times m$ sparse matrix is represented as an array of pointers to rows. The rows are arrays with a length followed by pairs of column indices and coefficients sorted in descending order. An overview of the elimination process is shown in Figure 1.

To perform row reduction, we use two auxiliary arrays of length m. The first, called *pivots*, is initialized to zero, and when we encounter a new pivot x_j which is the leading term of some reduced row r_i we assign $pivots_j := r_i$. This allows us look up a reducer for column j in constant time.

The second array is a dense buffer that is used to reduce rows. A row to be reduced is moved into the buffer, and we loop over the buffer contents in descending order, cancelling non-zero elements for which a pivot is known. If the result is not zero, we copy the buffer contents to a new sparse row, make it monic, and insert that row into the pivots array.

Our implementation uses 32-bit words for the rows of the matrix, 64-bit integers for the buffer, and row reduces mod a 31-bit prime p. To prevent overflow when subtracting $c \cdot \vec{a}$ from the buffer, we add p^2 if the value becomes negative, as shown in the listing of the innermost loop below. We found that using 32-bit integers produced a savings of around 5%,

Figure 1: Data structures for Gaussian elimination.



while unrolling the innermost loop below saved up to 34%. We attempted to vectorize the loop without success. Lastly, we tried replacing the % operator with multiplications but this had no effect as all of the time is in the innermost loop.

```
for (i=lt; i >= 0; i--) {
  c = buffer[i];
  if (c) c %= p;
  buffer[i] = c;
  if (!c) continue;

  a = pivots[i];
  if (a==NULL) continue;

  // subtract c times row a from the buffer mod p
  buffer[i] = 0;
  for (j=3; j < a[0]; j+=2) {
      t = buffer[a[j]];
      t -= c * a[j+1];
      t += (t >> 63) & p2; // t = (t<0) ? t+p2 : t;
      buffer[a[j]] = t;
  }
}
```

Several optimizations are important for the F4 algorithm. First, our implementation sorts the matrix so that rows are examined by increasing leading column. Secondly, we avoid reducing rows whose leading term is a new pivot – these are inserted directly into the pivots array, bypassing the buffer. In an earlier version of our code, we tried reducing all rows and re-using them in later matrices, but this never paid off. It also does not pay to perform only top reductions, that is, reducing rows just until a new pivot is found.

To parallelize this algorithm, each thread is given its own buffer to work in, and the reduction of each row is spawned as a *task* [2]. If the resulting row is non-zero, it is added to the pivots array using an atomic *compare and swap*. Should the CAS operation fail, the row is copied back to the buffer and reduced again, in which case we do more work than the sequential algorithm. Table 1 shows the total CPU time for sparse linear algebra as the number of threads increases, on an Intel Xeon E5-2680v2 2.8 GHz with 20 cores. On smaller problems there is a substantial increase in work due to this issue of extra work performed by threads under contention. On larger problems we obtain better speedups.

Table 1: CPU time and speedup for linear algebra.

CPU	1	2	4	8	12	16	20
cyclic-7	0.080	0.110	0.150	0.220	0.250	0.300	0.320
cyclic-8	1.920	2.080	2.260	2.500	2.910	3.060	3.210
cyclic-9	261.540	272.340	278.790	295.740	355.380	322.360	331.650
katsura-9	0.330	0.390	0.450	0.540	0.660	0.690	0.740
katsura-10	2.840	2.900	3.040	3.320	3.660	3.800	3.950
katsura-11	17.330	17.820	18.450	19.480	20.840	21.060	21.430
katsura-12	267.571	281.080	292.300	315.350	337.010	351.480	359.230

speedup	1	2	4	8	12	16	20
cyclic-7	1.00	1.54	2.22	3.20	3.64	4.00	4.21
cyclic-8	1.00	1.89	3.47	5.96	7.97	9.14	10.55
cyclic-9	1.00	1.92	3.75	7.07	10.02	12.98	15.76
katsura-9	1.00	1.76	3.20	5.50	5.89	6.73	7.86
katsura-10	1.00	1.96	3.73	6.83	9.44	11.74	13.27
katsura-11	1.00	1.94	3.74	7.07	9.97	13.10	16.06
katsura-12	1.00	1.90	3.66	6.78	9.50	12.18	14.89

2.3 Amdahl's Law

Table 2 shows a breakdown of the time spent in different parts of the algorithm for two larger examples. On cyclic-9, Gaussian elimination takes 85% of the total sequential time and the speedup is 15.76x. But on the entire Gröbner basis computation, the time drops from 305 to 60 seconds, which is only a speedup of 5x. The result for katsura-12 is more encouraging, the overall speedup is 8.58x. However, on this problem Gaussian elimination is 95% of the sequential time and its speedup is 14.89x. The times vary somewhat due to turbo boost, which allows the processor run faster provided the temperature stays low.

Table 2: Time breakdown for large Gröbner bases.

	cyclic-9		katsura-12	
	1 core	20 cores	1 core	20 cores
select pairs	7.560	7.322	4.159	4.163
symbolic preproc.	24.276	24.193	8.760	8.274
poly ↔ matrix	2.584	2.712	1.227	1.341
Gaussian elim.	261.540	16.596	267.571	17.972
update pairs	9.050	9.341	0.949	1.010
inter-reduce	0.257	0.290	0.170	0.205
total time	305.267	60.454	282.836	32.965

Our lack of speedup is explained by Amdahl's law, which states that for times $\{t_1, \ldots, t_n\}$ with speedups $\{s_1, \ldots, s_n\}$ the overall speedup is

$$\frac{t_1 + t_2 + \cdots + t_n}{t_1/s_1 + t_2/s_2 + \cdots + t_n/s_n}.$$

One can also see sequential costs taking a larger percentage of the total time when running in parallel with many cores.

Amdahl's law allows us to estimate the parallel speedup we can achieve by optimizing or parallelizing other parts of the program. For example, if we parallelize the construction of matrices in the algorithm (first two rows) and we achieve a speedup of 20x then we would expect the overall speedup on cyclic-9 to improve from 5x to 10x. This motivated us to try, and our results are reported in the next section.

An alternative would be to implement packed monomials as in Magma [16]. If we pack the exponents into bytes and use 64-bit arithmetic to multiply monomials then we might be able to gain up to a factor of eight. That would improve the parallel speedup on cyclic-9 from 5x to 9.3x. Note that whether we optimize or parallelize the effects are the same, so a small optimization to the sequential part of a program can have a large impact on parallel speedup.

2.4 Symbolic Preprocessing

After Gaussian elimination, the next largest cost in F4 is multiplying polynomials by monomials during the symbolic preprocessing step. In order to parallelize more, we have to first make our monomial operations threadsafe.

In the computation of cyclic-9, about 5% of all monomial operations produce a new monomial. Therefore, we decided to use a lock to protect the monomial block and hash table from parallel writes. The next problem was how to enlarge these structures if they fill up while other threads are using them in parallel. We did not attempt this. Instead, when a monomial operation has no storage available it simply fails. We detect this, acquire more space, and restart that part of the algorithm. In practice, we would enlarge the structures in advance to prevent failures from occurring. Our goal was to impose as little overhead as possible.

After several false starts we settled on a simple approach to parallelizing symbolic preprocessing. When a polynomial multiplication was to be performed, it would be spawned as a task [2]. In place of the result we would temporarily store a sentinel value, which would be overwritten on completion of the task. Cilk makes this very easy to do, e.g.:

```
a[i] = SENTINEL;
a[i] = cilk_spawn poly_mult(q, g);
```

If the symbolic preprocessing code encountered the sentinel it would issue a `cilk_sync;` and run all tasks to completion before proceeding. In practice this was very rare, suggesting the approach could be effective theoretically. We think this method of programming irregular parallelism is useful.

Unfortunately it did not work out so well, and the reason is interesting. Table 3 shows real and CPU time on cyclic-9 with parallel symbolic preprocessing on the right. Observe the explosion of CPU time for symbolic preprocessing when parallelism is used. We think this is due to lock contention. While it is true that 5% of monomial operations produce a new monomial, those operations are not evenly distributed; they are clustered near the start of symbolic preprocessing. This makes parallelization difficult, and we view the results in Table 3 as an unsuccessful experiment.

Table 3: Cyclic-9: parallel symbolic preprocessing.

	only Gauss elim.		symbolic preproc.	
	real	CPU	real	CPU
select pairs	7.598	7.598	0.708	13.770
symbolic preproc.	25.642	25.642	15.189	301.880
poly ↔ matrix	2.709	2.709	3.865	3.965
Gaussian elim.	16.754	334.430	17.269	344.940
update pairs	9.464	9.464	9.558	9.558
inter-reduce	0.302	0.302	0.308	0.308
total time	62.292	379.950	46.835	674.300

2.5 Rational Computations

The standard approach to computing Gröbner bases over the rationals is to use a modular method, which we present in the next section. For comparison, we implemented linear algebra using the fraction-free approach of [12], which does an order of magnitude less work on sparse problems. Sadly, the intermediate expression swell we observed suggests that direct computation of Gröbner bases over $\mathbb{Q}$ is not practical.

While reducing the buffer, the code maintains a common denominator d. It also stores a denominator for each buffer element. To cancel the leading term in the buffer, a/d, with

a row having leading coefficient b, we compute $g = \gcd(a, b)$, $d := db/g$, and $q = a/g$, and subtract q times the row from the buffer. The buffer entries that are modified by the row operation are put over the common denominator using one division and one multiplication if need be.

This method can save a significant amount of arithmetic if the buffer is long and the rows are short, since it updates only the entries colliding with a row, and at the same time avoids the costly gcds of rational arithmetic. The approach is efficient so long as the common denominator does not get too large. Unfortunately, this blowup occurs in the Gröbner basis computations we tested.

For example, cyclic-7 took 1 hour 40 minutes to compute. The result has coefficients up to 293 bits, but intermediate polynomials have coefficients up to 33431 bits. There is an acute blowup during zero-reductions, which used arithmetic with numbers approaching one million bits.

From this experiment we conclude that modular methods are critically important and we should reconstruct the final basis instead of the intermediate polynomials in each stage.

2.6 Multi-modular Computations

To compute Gröbner bases over the rationals we adopted a multi-modular approach. The top level algorithm written in Maple [11] calls our C library to compute Gröbner bases modulo a set of primes. Then we use Chinese remaindering and rational reconstruction to recover the basis over $\mathbb{Q}$. We implemented two methods: run our C library in serial with parallel Gaussian elimination, or run our library in parallel for several primes at once with serial Gaussian elimination. For the second method we spawned parallel processes using Maple's Grid package. The Grid package adds about 10 ms of overhead per call, which we consider acceptable.

Table 4 summarizes our results. We report the number of primes, the time for sequential Gröbner basis computation, the time for parallel computation with 20 threads, the time using Maple's Grid package which computed Gröbner bases in batches of 20 primes, the time for Chinese remaindering, and the time for rational reconstruction.

Table 4: Time breakdown of Gröbner bases over $\mathbb{Q}$.

	Gröbner bases mod p				Maple code	
	$\#p$	seq	par	grid	chrem	recon
cyclic-7	19	3.023	2.159	1.928	0.802	0.694
cyclic-8	44	120.801	48.844	15.446	6.552	2.712
katsura-9	27	15.352	4.517	10.041	3.796	2.604
katsura-10	43	176.571	35.340	65.485	36.867	10.527

Simply calling our parallel algorithm sequentially offers a speedup of 1.25x on cyclic-7 and 2.34x on cyclic-8, which is disappointing. For katsura-9 we obtain a speedup of 2x and for katsura-10 we obtain 2.7x.

The Grid method has a speedup of 5.26x on cyclic-8, but for katsura-9 and 10 the speedup *decreases* to 1.32x and 2x. We did not expect this. It turns out that reconstructing the basis in those cases creates a significant amount of garbage, and Maple's garbage collection costs increase substantially. This further slows down Chinese remaindering and rational reconstruction in Maple (not shown).

We tried varying the batch size and number of threads to perform fewer basis computations in parallel but with more parallelism in each one. We also tested a pre-release version of Maple 2015 which has a more efficient Grid package. The times for Gröbner bases modulo p are reported in Table 5.

Table 5: Varied numbers of threads and batch size.

threads/batch	20/1	10/2	5/4	2/10	1/20
cyclic-7	2.159	3.174	1.665	1.059	0.825
cyclic-8	48.844	32.267	21.639	16.176	16.267
katsura-9	4.517	6.044	4.782	2.956	4.493
katsura-10	35.340	37.201	28.316	crash	crash

Katsura-10 exposed a memory corruption bug which has been reported to Maplesoft.

While there is clearly an advantage to computing modulo batches of primes, there is also a disadvantage of increased memory use which could be prohibitive on larger examples. The (sequential) cost of Chinese remaindering and rational reconstruction is also a problem. For katsura-9 and 10 they limit parallel speedup to 3x and 4x respectively.

It is clear from Amdahl's law that we have to optimize or parallelize Chinese remaindering and rational reconstuction to obtain better speedup. In the meantime, our parallelized Gaussian elimination does a good job of reducing the total time spent for a modest number of cores, e.g. four, with no increase in space.

3. BENCHMARKS

We compared the performance of our software with FGb in Maple 18, Magma 2.21-2, and Singular 4-0-2, running on an Intel Xeon E5-2660 2.2 GHz (16 core) with 3 GHz turbo speed and 64 GB of RAM, running 64-bit Linux.

FGb and Magma use the F4 algorithm and Singular uses Buchberger's algorithm. We also tested the signature basis algorithm in Singular but its timings were similar.

Our first benchmark uses the short prime 32003, which is fast for all the systems tested. Our code handles primes up to $2^{31} - 1$ as does Singular. FGb needs a 16-bit prime, and Magma supports any size prime but is fastest for primes up to 11863279 (23.5 bits).

Table 6: Gröbner bases mod a short prime 32003.

	16 cores	4 cores	1 core	FGb	Magma	Singular
cyclic-7	0.135	0.146	0.197	0.194	0.140	1.330
cyclic-8	1.345	1.753	3.541	3.804	1.960	38.250
cyclic-9	73.853	139.710	411.466	433.137	166.730	–
katsura-9	0.184	0.295	0.715	1.198	0.380	8.540
katsura-10	0.964	1.850	5.356	9.608	2.540	64.030
katsura-11	6.013	14.019	45.801	76.794	18.710	685.260
katsura-12	44.481	112.127	383.921	677.874	630.010	–

The timings show that for one core we are comparable to FGb but not as fast as Magma. We attribute that to a lack of vectorization. With parallelization, which was fairly easy to do, we become the fastest program. We can expect good performance on a typical desktop machine with 4-8 cores.

The timings also show that the Buchberger algorithm can not compete with the F4 algorithm. It is worth considering why this is so. Recall that Buchberger's algorithm reduces S-polynomials one at a time. In doing so, it 1) searches the basis for divisors for each monomial, 2) multiplies elements of the basis by monomials, 3) subtracts polynomials using a merge that does monomial comparisons. The F4 algorithm amortizes all of these costs across a batch of pairs, and also largely eliminates storage management by making the inner loop operate in-place on a buffer.

Our next benchmark compares the systems by computing over the rationals. We use the parallel Gaussian elimination of Section 2.2, and reconstruct the final basis using Chinese remaindering and rational reconstruction. FGb implements

the same strategy (we think) but it is faster than our code. Magma is the fastest system of all. It seems to reconstruct the intermediate polynomials instead of just the final basis. The strategy seems to work well despite the blowup we saw in Section 2.5. Singular's Buchberger algorithm was unable to compete with the modular F4 algorithms in general.

Table 7: Gröbner bases over the rationals.

	16 cores	4 cores	1 core	FGb	Magma	Singular
cyclic-7	4.237	4.416	5.436	3.175	0.820	>6825.010
cyclic-8	73.362	92.371	171.070	87.476	18.420	–
katsura-9	15.184	17.936	30.176	13.717	2.700	140.970
katsura-10	105.110	144.155	298.283	101.494	21.370	>1688.580

We were surprised that our code is competitive modulo p yet we did not fair so well over the rationals. Clearly, there is more to do. One idea that we have not explored yet is to remember which rows reduce to zero in the computation for one prime and to avoid reducing those rows for subsequent primes. This process, called learning [14] could increase our throughput for computations over $\mathbb{Q}$. Another option is to vectorize the computation for several primes at once.

Our final benchmark is the cyclic-10 problem modulo p. This is an example of a very large computation that shows the benefits of parallelism. We tested Magma and our code with 16 cores, and we present a breakdown of the times for both systems. Both systems use 20GB of memory.

Table 8: Time breakdown for cyclic-10 benchmark.

	our library (16 cores)		Magma
	real time	CPU time	CPU time
select pairs	236.363	235.860	144.140
symbolic preproc.	1082.708	1080.780	440.780
poly $\leftrightarrow$ matrix	133.419	133.100	3.370
Gaussian elim.	5974.139	95331.660	31378.890
update pairs	446.000	445.260	113.030
inter-reduce	25.783	398.460	4.040
total time	7875.532	97229.960	32090.180

The table suggests several areas where we could improve. For example, by packing exponents into machine words the times for select pairs, symbolic preprocessing and updating the basis could be reduced. Still these optimizations, which may be time consuming, pale in comparison to parallelizing Gaussian elimination which takes the vast majority of time. Ultimately we were able to beat Magma by a factor of four using 16 cores, so with 4 cores we might expect a tie.

The lesson we think is that while there is value in having a highly optimized implementation, one should not overlook the easy gains available from parallelism. This will become especially obvious if the number of cores available on most machines continues to increase.

4. CONCLUSION

This paper presented a compact C implementation of the F4 algorithm for computing Gröbner bases modulo a prime. Our main contribution was a simple way to parallelize the Gaussian elimination. Other contributions are experiments in parallelizing symbolic preprocessing, an experiment with computing over the rationals directly, and an initial version of an algorithm for $\mathbb{Q}$ coded in Maple, which can compute modulo several primes in parallel.

From this we can see several options for future work. We plan to improve the algorithm for $\mathbb{Q}$, and it seems clear that to do so will require us to implement more things in C. We suspect there will be more opportunities for parallelism this way as well. For example, Chinese remaindering or rational reconstuction can operate on coefficients in parallel, and the gains could be significant as per Amdahl's law.

One question that we did not address here is whether our parallel Gaussian elimination remains efficient for matrices generated by the F5 algorithm, because those matrices have full rank. This would require an implementation of F5, and we think this would be an interesting research topic.

In any case we hope that our experience of implementing F4 with very little code encourages more computer algebra systems to adopt the algorithm. We also hope that an easy approach to parallelism, like the one presented here, allows more software to exploit the multicore computers of today.

With that in mind, a notable contribution is our method for parallelizing symbolic preprocessing. Although it didn't work out for us here, we think it is a good approach and we intend to try it in other settings.

5. REFERENCES

[1] B. Buchberger. Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory. *Multidimensional Systems Theory - Progress, Directions and Open Problems in Multidimensional Systems*, Reidel (1985), 184–232.

[2] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Proc. of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 207–216, 1995.

[3] C. Eder, J.C. Faugère. A survey on signature-based Gröbner basis computations. arXiv:1404.1774.

[4] J.C. Faugere. A new efficient algorithm for computing Gröbner bases (F4). *J. of Pure and Applied Algebra*, **139** (1999) 61–88.

[5] J.C. Faugère. A new efficient algorithm for computing Gröbnerr basis without reduction to 0 F5, *Proc. of ISSAC 2002*, ACM Press, (2002), 75–83.

[6] J.C. Faugère, S. Lachartre. Parallel Gaussian elimination for Gröbner bases computations in finite fields. *Proc. of PASCO 2010*, ACM Press, (2010), 89–97.

[7] R. Gebauer and H.M. Möller. On an Installation of Buchberger's Algorithm. *J. of Symb. Comp.*, **6** (2 and 3) (1988) 275–286.

[8] F. Hopgood and J. Davenport. The quadratic hash method when the table size is a power of 2. *The Computer Journal*, **15** (4) (1972) 314–315.

[9] H. Kredel. Distributed Parallel Groebner Bases Computation. *Proc. CISIS 2009*, IEEE, (2010), 518–524.

[10] A. Leykin. On parallel computation of Gröbner bases. *Proc. ICPP Workshops*, (2004), 160–164.

[11] Maple 2015. Maplesoft, a division of Waterloo Maple Inc., Waterloo, Ontario.

[12] M. Monagan, R. Pearce. Sparse Polynomial Division using Heaps. *J. Symb. Cmpt.* **46**(7) (2011), 807–822.

[13] S. Neumann. Parallel Reduction of Matrices in Gröbner Bases Computations. *Proc. of CASC 2012*, Springer, (2012), 260–270.

[14] B. Parisse. A probabilistic and deterministic modular algorithm for computing Groebner basis over Q. arXiv:1309.4044.

[15] A. Reeves. A parallel implementation of Buchberger's algorithm over Zp for p $\leq$ 31991. *J. Symb. Comp.* **26** (1998) 229–244.

[16] Allan Steel. Private communication.

[17] J.P. Vidal. The computation of Gröbner bases on a shared memory multiprocessor. *LNCS 429*, Springer, (1990), 81–90.