Sparse Polynomial Powering Using Heaps

Michael Monagan and Roman Pearce

Department of Mathematics, Simon Fraser University, Burnaby, B.C., Canada
mmonagan@cecm.sfu.ca and rpearcea@cecm.sfu.ca

Abstract. We modify an old algorithm for expanding powers of dense polynomials to make it work for sparse polynomials, by using a heap to sort monomials. It has better complexity and lower space requirements than other sparse powering algorithms for dense polynomials. We show how to parallelize the method, and compare its performance on a series of benchmark problems to other methods and the Magma and Singular computer algebra systems.

Key words: Sparse Polynomials, Powers, Heaps, Parallel Algorithms.

1 Introduction

Expanding powers of sparse polynomials is an elementary function of computer algebra systems. Despite receiving a lot of attention in the 1970's, a fragmented situation exists today where the fastest sparse methods make time and memory tradeoffs that improve one case at the expense of others. Thus, programmers of computer algebra systems must implement multiple routines and carefully select among them to obtain good performance.

For an introduction to this problem and current methods it is hard improve on the papers by Richard Fateman [1, 2]. He characterizes the relative performance of the algorithms by counting coefficient operations. We briefly discuss these results. Let f be a polynomial with t terms to be raised to a power $k > 1$. We use f_i to refer to the ith term of f and $\#f$ to refer to the number of terms of f. We consider two cases: *sparse* and *dense*.

In the sparse case, the terms of f interact as if they were algebraically independent, e.g. as in $f = x_1 + x_2 + \cdots + x_t$. Expanding f^k creates $\binom{k+t-1}{k}$ terms, the most possible. In the dense case the terms of f combine as much as possible, e.g. as in $f = 1 + x + x^2 + \cdots + x^{t-1}$. If there are no cancellations, f^k will have $k(t-1) + 1$ terms.

We want a sparse algorithm to have good performance in the dense case, to allow for a smooth transition to dense methods inside a general purpose routine. The literature suggested that current sparse methods do an order of magnitude too much work in the dense case, so we developed new methods to address this. This in turn forced us to reassess sparse and dense algorithms for powering, as the consensus heavily favors dense algorithms.

Our contribution is two methods for powering sparse polynomials. The first, Sparse SUMS, has the best performance in the dense case. The second method, which we call FPS, is a modification to improve performance in the sparse case.

Let us review the methods in the literature.

RMUL computes $f^i = f \cdot f^{i-1}$ for $i = 2 \ldots k$. The memory taken by f^{i-2} may be reused to hold f^i so that total storage is at most twice the result.

RSQR computes $f^i = (f^{i/2})^2$ for $i = 2 \ldots \lfloor \log_2 k \rfloor$, with extra multiplication by f at each 1 in the binary expansion of k. E.g. $f^{13} = f^{1101_2} = (((f)^2 \cdot f)^2)^2 \cdot f$.

Gentleman and Heindel note in [4, 5] that *RSQR* is vastly inferior to *RMUL* in the sparse case. *RSQR* also requires asymptotically fast dense multiplication to improve on *RMUL* in the dense case. Therefore, *RSQR* is a dense algorithm. The best feature of *RMUL* is that it aggressively combines like terms. This can be of great importance on large problems which "fill-in". Its weakness is sparse problems and high powers.

BINA selects $f_1 \in f$ and expands $g = (f_1 + 1)^k$ using the binomial theorem. It expands $(f - f_1)^i$ for $i = 2...k$ using *RMUL* and merges $f^k = \sum_{i=0}^{k} g_i \cdot (f - f_1)^i$.

BINB is similar to *BINA* except that f is split into equal-sized parts $f = g + h$. It expands and merges $f^k = \sum_{i=0}^{k} \binom{k}{i} \cdot g^i \cdot h^{k-i}$.

Binomial methods originate with Fateman in [1], who shows that *BINB* is nearly optimal in the sparse case. Alagar and Probst [11] improve on this using recursion, and Rowan [16] expands the set of powers $\{g^i\}$ more efficiently, both for the sparse case only. For the dense case, Fateman in [2] shows that *BINA* is comparable to *RMUL* and much faster than *BINB*. The tradeoff made in *BINB* assumes that few like terms combine. This makes it unsuitable for our purpose. In *BINA*, we avoid unbalanced merging by storing all $(f - f_1)^i$ and performing a simultaneous n-ary merge that multiplies by each g_i inline. This makes *BINA* extremely fast in most cases, at the cost of extra memory.

MNE generates all combinations of terms with multinomial coefficients, see [6]. This quickly becomes infeasible in the dense case.

FFT performs fast multipoint evaluation at roots of unity modulo primes, uses modular exponentiation on the values, then performs fast interpolation. Over $\mathbb{Z}$ it uses multiple primes and Chinese remaindering.

As noted by Ponder in [10], the FFT can be competitive in practice because high powers of sparse polynomials tend to fill in. For multivariate polynomials, one can use the Kronecker substitution as suggested by Moenck [9], however this separates the variables with very high degrees and thus limits gains from fill-in. A weakness of the FFT is that small polynomials raised to high powers over $\mathbb{Z}$ require many large FFTs. For that case the following classical method is faster, a crucial fact which was brought to our attention by Greg Fee.

SUMS is a dense method. Let $f = \sum_{i=0}^{d} f_i x^i$. To compute $g = f^k = \sum_{i=0}^{kd} g_i x^i$ we compute $g_0 = f_0^k$ and use the formula $g_i = \frac{1}{i f_0} \sum_{j=1}^{\min(d,i)} ((k+1)j - i) f_j g_{i-j}$ for $i = 1 \ldots kd$.

The *SUMS* algorithm is originally due to Euler and is used to exponentiate power series, see [2, 3, 8]. The algorithm is extremely fast for small polynomials raised to large powers, as it is linear in k and quadratic in d.

Two features of the *SUMS* formula recall the sparse multiplication algorithm of Johnson [7]. First, it computes each new term of the result in order. Second, it merges pairwise products $f_j g_{i-j}$ of equal degree, but scaled by $((k+1)j-i)$. Our starting point was to make a sparse method by skipping over products that a sparse representation omits, that is, where f_j or g_{i-j} equals zero.

What methods do computer algebra systems presently use for this problem? Singular 3.1 uses *RMUL*. Magma 2.17 uses *RSQR*. Maple 16 selects among our implementations of *RMUL*, *BINA*, and *RSQR*. For univariate powering, Maple estimates when *RSQR* will beat *BINA*. For multivariate powers, Maple bounds the extra memory needed for *BINA* and uses *RMUL* when this is too large.

For the underlying multiplications, Magma and Maple use dense algorithms for univariate polynomials over $\mathbb{Z}$. Magma uses the Schönhage-Strassen method with a single modulus of the form $2^{2^k}+1$. Maple evaluates at a large integer of the form 2^{64i} to leverage the FFT from integer multiplication. For multivariate multiplications, Maple, Magma, and Singular all use classical sparse algorithms and distributed polynomial representations. Maple uses our codes from [12, 14].

Our paper is organized as follows. Section 2 develops the Sparse SUMS and FPS algorithms and describes our implementation. The complexity of powering is discussed in Section 2.1. Section 2.2 describes our approach to parallelization which we also used successfully for sparse polynomial division in [15]. Section 3 compares the performance of the algorithms on benchmark problems.

2 Sparse Sums

For completeness we briefly derive *SUMS*. Let $f = \sum_{i=0}^{d} f_i x^i \in \mathbb{Q}[x]$ and $g = f^k$. Then $g' = k\, f^{k-1} \cdot f'$ and $f \cdot g' = k\, g \cdot f'$. Equating terms of degree $i-1$ in

$$(f_0 + f_1 x + \cdots)(g_1 + 2 g_2 x + \cdots) \;=\; k(g_0 + g_1 x + \cdots)(f_1 + 2 f_2 x + \cdots)$$

we obtain

$$\sum_{j=0}^{\min(d,i)} f_j x^j \cdot (g_{i-j} x^{i-j})' = \sum_{j=1}^{\min(d,i)} k g_{i-j} x^{i-j} \cdot (f_j x^j)'$$

from which we isolate g_i to obtain the formula for $i > 0$. □

Algorithm: Dense SUMS (descending order).

Input: dense polynomial $f = f_0 + f_1 x + \cdots + f_d x^d$, $f_d \neq 0$ stored as an array $[f_0, f_1, \ldots, f_d]$ indexed from zero, and a positive integer k.

Output: dense polynomial $g = f^k$.

1 $g :=$ an array with $kd+1$ elements indexed from zero
2 $g_{kd} := f_d^k$
3 for i from $kd-1$ to 0 by -1 do
4 $\quad e := kd - i$
5 $\quad c := \sum_{j=1}^{\min(d,e)} ((k+1)j - e) \cdot f_{d-j} \cdot g_{i+j}$
6 $\quad g_i := c/(e \cdot f_d)$
7 return g

Our first task is to modify *SUMS* to produce the terms in descending order, dividing by the leading coefficient of f rather than the constant term f_0. This leads into the sparse version and solves the problem of what to do when $f_0 = 0$.

In algorithm Dense SUMS we identify i as the degree of the next term being computed for g. To compute g_i, we merge products of degree $i + d$, scaling by $((k+1)j - e)$. To make our sparse algorithm, we express this scale factor using the terms' degrees. To merge $f_\alpha x^\alpha \times g_\beta x^\beta$ where $\alpha + \beta = i + d$, we scale by $((k+1)j - e) = \beta - k\alpha$.

The sparse version of *SUMS* is presented below. It uses a heap of pointers into f and g to combine only the products $f_i \times g_j$. The idea is to use a heap to merge the set of all pairwise products $f_i \times g_j$ in descending order. A property of $X+Y$ sorts, namely that $f_i \times g_j$ is strictly greater than $f_i \times g_{j+1}$ and $f_{i+1} \times g_j$, is used to reduce the number of products compared in the heap where possible. This optimization is fully exploited in our other heap based routines [12, 14, 15]. Also note, because the coefficients of g are much larger than those of f, there is an advantage to multiplying $(\beta - k\alpha) \cdot cof(f_i)$ first.

Algorithm: Sparse SUMS.

Input: sparse univariate polynomial $f = f_1 + f_2 + \cdots + f_t \in \mathbb{Z}[x]$
with terms descending in degree, and a positive integer k.

Output: sparse polynomial $g = f^k$.

1 $H :=$ an empty heap ordered by degree with maximum element H_1
2 $g := f_1^k$
3 insert $f_2 \times g_1 = (2, 1, \deg(f_2) + \deg(g_1))$ into H
4 while $|H| > 0$ and $\deg(H_1) \geq \deg(f)$ do
5 $M := \deg(H_1)$; $C := 0$; $Q := \{\}$;
6 while $|H| > 0$ and $\deg(H_1) = M$ do
7 $(i, j, M) := extract_max(H)$
8 $(\alpha, \beta) := (degree(f_i), degree(g_j))$
9 $C := C + (\beta - k\alpha) \cdot cof(f_i) \cdot cof(g_j)$
10 $Q := Q \cup \{(i, j)\}$
11 for all $(i, j) \in Q$ do
12 if $j < \#g$ and ($i = 1$ or $f_{i-1} \times g_{j+1}$ was merged) insert $f_i \times g_{j+1}$ into H
13 if $i < \#f$ and $f_{i+1} \times g$ not in H then insert $f_{i+1} \times g_j$ into H
14 if $C \neq 0$ then
15 $C := C/((\deg(g_1) - M) \cdot cof(f_1))$
16 $g := g + C\, x^{M - \deg(f_1)}$
17 if $f_2 \times g$ has no term in H then insert $f_2 \times g_{\#g}$ into H
18 return g

In computer memory, the heap is an array of size $O(\#f)$ with pointers into a second array for the products $f_i \times g_j$. For most inputs (1000 terms or fewer) these structures fit inside the L1 cache. For each $f_i \in f$, we maintain a pointer to the next term $g_j \in g$ for which we have yet to merge $f_i \times g_j$. This makes the test for whether $f_{i-1} \times g_j$ has been merged easy. We simply check if the pointer for f_{i-1} has advanced beyond g_j. We set a bit to indicate whether each product $f_i \times g_j$ is in the heap or not.

2.1 Complexity and FPS

Theorem 1. *Sparse sums expands $g = f^k \in \mathbb{Z}[x]$ using $(2\,\#f - 1)\,\#g + 2\log k$ coefficient multiplications, $\#g$ divisions, and $O(\#f\#g\log\#f)$ comparisons.*

Proof. Binary powering $g_1 = f_1^k$ does at most $2\log k$ multiplications. We merge the set of all products $\{f_i \times g_j\}$ for $2 \le i \le \#f$ and $1 \le j \le \#g$ with the heap. Each product requires two multiplications in line 9 and $O(\log\#f)$ comparisons for the heap in lines 7, 10 and 13. We do not count the exponent multiplication in $\beta - k\alpha$. To construct each term of g, we perform one multiplication and one division in line 15. □

For dense polynomials the $O(\#f\#g\log\#f)$ comparisons for the heap are a bottleneck. We use a heap optimization called *chaining* in all of our algorithms that reduces the cost of heap operations to $O(1)$ in the dense case. See [13, 14]. This optimization provides large gains on most problems.

For multivariate polynomials we use the Kronecker substitution to treat the problem as univariate. In general, one can use any invertible map of monomials to integers so long as monomial multiplications correspond to integer additions. The mapping has two caveats that we have not seen in other sparse algorithms. Because we multiply by the exponents, any padding in the map that increases the univariate degrees can also increase the cost of arithmetic in Sparse SUMS. And, because we divide by the exponents, we can not run the algorithm mod p if the degree of g under the mapping is greater than or equal to p.

Our benchmarks revealed one case where *Sparse SUMS* is highly inefficient. For low powers of extremely sparse polynomials, the set of monomials $\{f_i \times g_j\}$ is much larger than $\#g$, and the algorithm spends a substantial amount of time computing zero. On multivariate problems we can skip monomials not divisible by f_1, but the resulting improvement is often modest.

Instead, we observe that *Sparse SUMS* could construct f^{k+1} almost for free because it already multiplies every term of $g = f^k$ by every term of f except f_1. To exploit that fact, we created a variant of the algorithm that computes f^{k-1} and outputs f^k as a side effect. We call this alternative method *FPS*.

Table 1. Coefficient multiplications to power (t terms)^k.

	sparse case	dense case
RMUL	$\frac{(k+t-1)!}{(t-1)!(k-1)!} - t$	$t(k-1)(kt-k+2)/2 \in O(k^2t^2)$
BINA	$\frac{t\cdot(k+t-2)!}{(t-1)!(k-1)!} + 2k$	$t(k-1)(kt-2k+4)/2+2 \in O(k^2t^2)$
BINB	$\frac{(k+t-1)!}{k!(t-1)!} + \ldots$	$k^2(k-1)(t-2)^2/24 + \cdots \in O(k^3t^2)$
SUMS	$\frac{(2t-1)(k+t-1)!}{k!(t-1)!}$	$(2t-1)((t-1)k+1) \in O(kt^2)$
FPS	$\frac{(2t-1)(k+t-2)!}{(k-1)!(t-1)!}$	$(2t-1)((t-1)(k-1)+1) \in O(kt^2)$

Table 1 counts coefficient multiplications to compare the cost of algorithms. The sparse result has $(k+t-1)!/(k!(t-1)!)$ terms, so *BINB* is nearly optimal. *RMUL* is more expensive by a factor of k, slowing it down on high powers, and *BINA* by a factor of $kt/(k+t-1)$, which balances contributions from k and t. *Sparse SUMS* adds a factor of $(2t-1)$ and *FPS* a factor of $(2t-1)k/(k+t-1)$. Those methods also do divisions, which matter here but do not dominate.

The FFT is inefficient for sparse problems. One may assume these problems have distinct variables, e.g. $(1+x+y+z)^{50}$, and Kronecker substitution must separate variables in the result. For t terms to the power k, we must replace the ith term by at least $x^{(k+1)^{i-2}}$ for $i>2$, so the degree of f^k is $d=k\,(k+1)^{t-2}$.

An FFT does about $\frac{1}{2}n\log_2 n$ multiplications, where n is the first power of 2 greater than d. For example, $(1+x+y+z)^{50}$ will have $d=50\cdot 51^2=130050$ and $n=2^{14}$. The two FFT calls do about $n\log_2 n = 2.29\times 10^6$ multiplications, but *SUMS* and *FPS* require 1.64×10^5 and 1.55×10^5 operations respectively. In the dense case, *SUMS* and *FPS* are $O(kt^2)$ and the other sparse algorithms are $O(k^2t^2)$. Only the FFT can beat them, for sufficiently large polynomials.

We present *FPS* below, by simply adding lines to the description of *SUMS*. To lower the cost, $cof(f_i)\cdot cof(g_j)$ can be reused in lines 9 and 11, and in lines 17 and 18 we can compute $C := C/(\deg(g_1)-M)$; $S := S+C$; $C := C/cof(f_1)$.

Algorithm: FPS.

Input: sparse univariate polynomial $f=f_1+f_2+\cdots+f_t\in\mathbb{Z}[x]$ with terms descending in degree, and a positive integer k.
Output: sparse polynomial $h=f^k$.

```
1  H := an empty heap ordered by degree with maximum element H_1
2  g := f_1^{k-1};   h := f_1^k
3  insert f_2 × g_1 = (2, 1, deg(f_2) + deg(g_1)) into H
4  while |H| > 0 do
5    M := deg(H_1); C := 0; S := 0; Q := {};
6    while |H| > 0 and deg(H_1) = M do
7      (i, j, M) := extract_max(H)
8      (α, β) := (degree(f_i), degree(g_j))
9      S := S + cof(f_i) · cof(g_j)
10     if M ≥ deg(f_1) and β ≠ (k − 1)α then
11       C := C + (β − (k − 1)α) · cof(f_i) · cof(g_j)
12     Q := Q ∪ {(i, j)}
13   for all (i, j) ∈ Q do
14     if j < #g and (i = 1 or f_{i−1} × g_{j+1} was merged) insert f_i × g_{j+1} into H
15     if i < #f and f_{i+1} × g not in H then insert f_{i+1} × g_j into H
16   if C ≠ 0 then
17     C := C/((deg(g_1) − M) · cof(f_1))
18     S := S + C · cof(f_1)
19     g := g + C x^{M−deg(f_1)}
20     if f_2 × g has no term in H then insert f_2 × g_{#g} into H
21   if S ≠ 0 then
22     h := h + S x^M
23 return h
```

2.2 Parallelization

Our design for the parallel algorithm follows the approach used for polynomial division in [15]. Both problems have a tight data-dependency among the terms in the result. That is, each new term of g can depend on any subset of previous terms with no predictable pattern. To create parallelism we split the work into dynamically interacting pieces and exploit structure to hide latencies.

Fig. 1. Threads multiply strips of f by all of g. A global function merges the results from the threads and the first strip, while computing new terms of g.

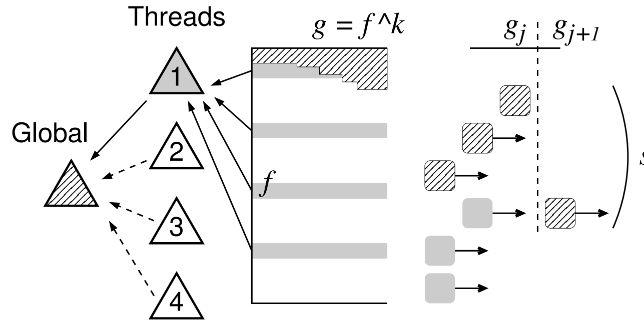

Figure 1 shows features common to all our parallel algorithms. The work of merging products $f_i \times g_j$ is divided into strips along the terms of f, so threads are given subsets of f to multiply by g. A global function combines their results and computes new terms of g. This function is protected by a lock and may be called by any thread, which allows them to cooperatively balance the load [12].

Another feature from our earlier work on division [15] is used to resolve the data-dependency. The first strip of f is assigned to the global function, so that as new terms g_j are computed there is no delay in merging $f_2 \times g_j$. Recall that this term must be compared to all others immediately as it could be used next.

The global strip is also used to resolve the nasty problem of blocked threads. Threads block when they merge $f_i \times g_j$ and go to insert $f_i \times g_{j+1}$ in their heap only to find that g_{j+1} does not exist. The reason could be a delay, but perhaps $f_i \times g_j$ was merged by the global function and no new term of g was computed. In that case, the global function now needs $f_{i+1} \times g_j$ to progress. Our solution is for the global function to steal rows from the threads when this happens.

To implement stealing, we have two shared variables that are read by all of the threads. The first variable t is the number of terms computed in the result. The variable s is the number of rows stolen by the global function. To ensure a valid state, threads must read s before t, and the global function must update t before incrementing s. We enforce this with memory barriers.

Incrementing t means that a new term of g was computed, and alongside its monomial and coefficient the global function stores the current value of s. This tells the threads what products involving g_t are stolen and must not be merged. When threads block waiting for t to be incremented, they attempt to enter the global function and then they update their local copies of s and t. The global function can steal rows with impunity. We do this whenever it is blocked.

Table 2. Timings (in CPU seconds) for completely sparse (t terms)^k.

input		result			C code			Magma RSQR		Singular
t	k	*terms*	*degree*	*bits*	*SUMS*	*RMUL*	*BINA*	*FFT*	*sparse*	*RMUL*
3	100	5151	10100	152	0.001	0.030	0.001	0.010	0.250	0.050
3	250	31626	62750	388	0.010	0.480	0.010	0.450	12.840	1.040
3	500	125751	250500	784	0.050	4.570	0.050	3.480	278.13	12.750
3	1000	501501	1001000	1575	0.320	45.630	0.290	31.380	–	125.29
3	2500	3128751	6252500	3951	4.130	–	5.260	(*)	–	–
4	50	23426	130050	92	0.005	0.030	0.005	0.120	1.340	0.180
4	100	176851	1020100	191	0.060	0.760	0.060	3.100	98.710	2.490
4	200	1373701	8080200	389	0.480	13.308	0.480	74.360	–	44.610
4	400	10827401	64320400	788	5.180	252.23	5.160	–	–	889.79
5	40	135751	2756840	83	0.030	0.130	0.020	4.300	13.890	0.680
5	60	635376	13618860	128	0.180	0.580	0.130	38.040	371.76	5.300
5	80	1929501	42515280	174	0.580	6.460	0.460	171.46	–	21.860
5	100	4598126	103030100	220	1.530	19.950	1.280	–	–	66.810
5	140	17178876	392450940	312	6.320	110.34	5.190	–	–	444.74
6	20	53130	3889620	42	0.010	0.030	0.010	3.250	0.770	0.070
6	30	324632	27705630	67	0.060	0.190	0.040	63.270	26.000	1.170
6	40	1221759	113030440	91	0.370	1.500	0.220	–	460.42	6.670
6	50	3478761	338260050	117	1.150	6.670	0.810	–	–	26.890
6	70	17259390	1778817670	167	6.050	51.140	4.840	–	–	176.80
8	15	170544	251658240	34	0.030	0.050	0.020	(*)	0.950	0.100
8	20	888030	1715322420	47	0.170	0.310	0.130	–	36.200	1.840
8	25	3365856	7722894400	62	0.640	1.430	0.460	–	284.64	10.700
8	30	10295472	2.66×10^{10}	76	2.780	5.750	1.530	–	–	42.920
8	35	26978328	7.62×10^{10}	90	9.580	26.800	5.680	–	–	148.97
12	10	352716	2.59×10^{11}	22	0.090	0.070	0.050	–	1.600	0.180
12	12	1352078	1.65×10^{12}	29	0.340	0.310	0.170	–	11.850	0.890
12	14	4457400	8.07×10^{12}	35	1.130	1.110	0.600	–	78.810	4.060
12	16	13037895	3.22×10^{13}	41	3.090	3.440	1.720	–	500.20	21.990
12	18	34597290	1.10×10^{14}	47	8.510	10.410	4.830	–	–	(*)

– Not attempted. (*) Ran out of memory.

3 Benchmarks

Our benchmarks were performed on a 2.66 GHz Intel Core i7 920 with 6 GB of RAM running Linux. This is a 64 bit 4 core processor. Timings are the median time in seconds of 3 runs. Magma is version 2.17 while Singular is version 3.10. Timings for *SUMS*, *RMUL*, and *BINA* are from our C library.

3.1 Sparse Problems

To create polynomials with t terms whose powers up to k are completely sparse, we may use Kronecker's substitution on $F = 1 + x_1 + x_2 + \cdots + x_{t-1}$ to construct

$$f = 1 + x + x^{(k+1)} + x^{(k+1)^2} + \cdots + x^{(k+1)^{t-2}}.$$

This polynomial to the power k generates the largest possible number of terms. That is what is meant by sparse. Notice how we can not have too many terms t before the integer exponents become massive. This suggests that most practical problems (whose result can be stored) have $t \ll k$, so the extra factor of $2t-1$ in the cost of sparse *SUMS* is not as disadvantageous as it may first appear.

In Table 2 we compare our *SUMS* method to *RMUL* and *BINA*. The polynomials are too short to run our parallel routines. For Magma we give two times; FFT is the binary algorithm for univariate powering with Schönhage-Strassen multiplication, but we also tried writing the problem as multivariate and using Magma's sparse *RSQR*. Singular uses *RMUL* which is a sensible choice.

The timing data shows that on sparse problems *SUMS* is consistently better than *RMUL* and almost as fast as *BINA*. Note that for *BINA* to run as fast as it does here it must store the polynomials $(f - f_1)^i$ for $1 \le i \le k$ and do a simultaneous n-ary merge. This doubles the space versus *SUMS*, which stores a tiny heap and the result.

3.2 Dense Problems

Table 3 shows timings for expanding powers of the polynomial

$$f = 1 + x + x^2 + \cdots + x^{t-1}.$$

Timings for *SUMS* for $t = 500$ and $t = 1000$ are real timings on 1 core and 4 cores. For $t = 500$ our code runs 3 threads so the speedup is a factor of 3 not 4. For $t = 1000$ it runs 4 threads and the speedup approaches 4.

Dense problems are a strong case for *SUMS*. *RMUL* and *BINA* are competitive only for low powers. Higher powers benefit *SUMS* even versus the FFT. For 500 terms, *SUMS* goes from 14 times slower than the FFT at k=20 down to 1.5 times slower for k=320 and breaks even at k=640. For a sparse algorithm this is a good result. *SUMS* dominates the timings for t=10 and t=100 terms.

Table 3. Timings (in CPU seconds) for completely dense (t terms)^k.

t	k	*SUMS*	*RMUL*	*BINA*	*FFT*	t	k	*SUMS*	(4 cores)	*RMUL*	*FFT*
10	200	0.000	0.085	0.095	0.006	500	10	0.0849	0.0313	0.150	0.004
10	500	0.005	0.760	1.035	0.095	500	20	0.2005	0.0716	1.330	0.014
10	1000	0.020	4.450	7.930	0.501	500	40	0.4734	0.1638	6.955	0.057
10	1500	0.040	13.370	28.950	0.510	500	80	1.1889	0.4165	35.075	0.247
10	2000	0.054	29.811	–	2.640	500	160	3.4083	1.1575	192.326	1.352
10	2500	0.086	55.749	–	2.670	500	320	10.6490	3.5878	–	6.890
100	50	0.020	0.420	0.420	0.017	1000	3	0.0432	0.01452	0.035	0.001
100	100	0.055	2.090	2.110	0.056	1000	5	0.0750	0.02298	0.115	0.003
100	200	0.155	11.090	11.425	0.262	1000	10	0.3633	0.09679	0.785	0.013
100	400	0.490	65.980	69.565	1.360	1000	20	0.8324	0.21790	5.735	0.030
100	800	1.700	439.380	–	6.990	1000	40	1.9637	0.50269	29.250	0.130
100	1600	6.156	–	–	36.310	1000	80	4.6245	1.28255	148.835	0.570

Table 4 considers powers of two dense multivariate polynomials. The data shows the sparse methods beat the FFT as the number of variables increase even though the polynomials are dense which favors the FFT. *SUMS* beats the other sparse methods for large k but not for small k. This is because for multivariate f and small k, we often have $\#f^k \gg \#f^{(k-1)} \gg ... \gg \#f^3 \gg \#f^2$. For such cases *RMUL* does just over $t\#f^{(k-1)}$ coefficient multiplications but *SUMS* does $2t\#f^k$. We find that for small t parallel speedup for *SUMS* is limited and drops off as k increases. For larger t parallel speedup improves.

Table 4. Timings (in CPU seconds) for dense multivariate f^k.

$f = (1+x+y)^{15}$ $t = 136$								Magma	Singular
k	$\#g$	SUMS (4 cores)		RMUL (4 cores)		BINA (4 cores)		FFT	RMUL
20	45451	0.537	0.151	1.513	0.421	1.554	0.439	0.49	12.33
40	180901	3.167	0.848	15.810	4.214	16.018	4.444	5.49	134.59
60	406351	9.303	2.484	65.342	17.085	65.500	17.996	27.27	522.59
80	721801	20.765	5.399	183.694	47.406	185.721	50.943	56.42	–
120	1622701	60.237	15.809	–	–	–	–	325.60	–
$f = (1+w+x+y+z)^4$ $t = 70$								Magma	Singular
k	$\#g$	SUMS (2 cores)		RMUL (2 cores)		BINA (2 cores)		FFT	RMUL
4	4845	0.0075	0.0075	0.0027	0.0022	0.0029	0.0024	0.30	0.01
8	58905	0.0663	0.0588	0.0671	0.0457	0.0679	0.0480	1.24	1.01
12	270725	0.6796	0.4228	0.9034	0.5708	0.9103	0.5837	10.84	10.40
16	814385	2.2402	1.3432	4.9317	2.9980	4.9729	3.0815	65.50	46.49
20	1929501	5.7590	3.4088	16.2635	9.7505	16.3970	9.8145	218.14	166.02
24	3921225	11.8027	8.6765	41.9719	24.5222	42.1545	24.8397	391.42	394.08
28	7160245	22.5105	21.7034	92.1249	53.6990	92.6547	54.5460	(*)	–

3.3 Real Examples

We were first motivated to investigate sparse powering by a post to the Sage development newsgroup by Tom Coates. He wanted to raise the polynomial

$$f = xy^3z^2 + x^2y^2z + xy^3z + xy^2z^2 + y^3z^2 + y^3z$$
$$+2y^2z^2 + 2xyz + y^2z + yz^2 + y^2 + 2yz + z$$

to high powers but no computer algebra system could do so in a reasonable amount of time. This can now be done quickly. Table 5 shows that *SUMS* is by far the best method. Note, in order to get Magma to use the FFT, we explicitly converted $f(x,y,z)$ into a univariate polynomial using Kronecker's substitution. Otherwise Magma uses sparse *RSQR* which is not competitive on this problem; it takes 134.49s for $k = 40$.

In [17] Zeilberger writes (in 1994)

> "In my research on constant term conjectures, I often need to expand powers of polynomials P^m where m is very large and P is (usually) a polynomial of several variables. I was frustrated by the slowness of all the commercial computer algebra packages. For example, in Maple, it takes several days to expand $(1 + 3x + 2x^2)^{3000}$."

Table 5. Timings (in CPU seconds) to power f^k.

		C code			Magma	Maple	Singular
k	$\#g$	*SUMS*	*RMUL*	*BINA*	*FFT*	*RMUL*	*RMUL*
40	243581	0.159	0.968	0.941	1.47	1.36	5.50
70	1284816	0.941	10.833	10.624	28.26	13.97	62.85
100	3721951	3.026	48.932	51.670	93.64	59.37	316.11
150	12499176	10.880	276.320	–	(*)	324.00	–
250	57636126	68.626	–	–	–	–	–

– Not attempted. (*) Ran out of memory.

Zeilberger coded dense *SUMS* in Maple and noted that it was theoretically faster than the FFT though his analysis does not take into account the size of the integers in the result which grow to over 2,300 digits long in his example.

At that time (1994) Maple was using *BINA* which is a bad choice here as it needs over 2 gigabytes to store the expanded powers of $(3x+1)^i$ for $0 \leq i \leq 3000$. Maple 15 and 16 use *RSQR* with the univariate polynomial multiplications done by evaluating at a large integer to leverage the FFT from fast integer multiplication. Maple 15 and 16 take 1 second on our Intel Core i7 @ 2.66 GHz computer. However *SUMS* takes less than 9 milli-seconds! It does less than $2t^2k = 2 \times 9 \times 3000$ coefficient multiplications. Actually, since the coefficients of $f = 2x^2 + 3x + 1$ are small, at most half of these multiplications (see line 9 of Sparse *SUMS*) are multi-precision. Column digits shows the length in decimal digits of the largest coefficient in the output.

Table 6. Timings (in CPU seconds) to power $(2x^2 + 3x + 1)^k$.

		C code			Magma	Maple	Singular
k	digits	*SUMS*	*RMUL*	*BINA*	*FFT*	*RSQR*	*RMUL*
1000	777	0.00130	0.302	0.591	0.02	0.088	0.76
2000	1555	0.00418	1.858	6.562	0.08	0.419	4.62
3000	2333	0.00884	5.461	28.847	0.25	1.03	15.04
4000	3111	0.01540	12.202	83.870	0.41	2.13	35.57
5000	3889	0.02318	23.008	(*)	1.31	3.48	70.32

(*) *BINA* ran out of space; it exceeded the 6 gigabytes available. Note *BINA* needs to expand and store the powers $(3x+1)^i$ for $0 \leq i \leq k$.

4 Conclusion

We adapted a classical method for powering dense series to make a new method for powering sparse polynomials. *SUMS* has better complexity than other sparse algorithms in the dense case, which is important for general problems. It has reasonable performance in the completely sparse case.

In comparing *SUMS* with *RMUL*, the larger the power and the smaller the polynomial, the better. We also compared it to the FFT and explained why the FFT struggles to power multivariate polynomials. It is due to the very high degrees that are needed in Kronecker substitution when powering. We conclude

that *SUMS* has a wide range of applicability. It performed extremely well on a benchmark problem coming from a real application.

Our effort to parallelize Sparse *SUMS* was largely successful. For inputs with a large number of terms, 500 or more, we often obtained good parallel speedup. A problem with this approach is that it requires the input to have a lot of terms, at least 50, to conceal communication latencies.

Our next task is to optimize and parallelize the *FPS* variant presented here. That algorithm should offer better performance in the cases where *SUMS* loses to *RMUL* or *BINA*, while retaining the best qualities of *SUMS*.

References

1. R. Fateman. On the computation of powers of sparse polynomials. Studies in Appl. Math., 53 (1974), pp. 145–155.
2. R. Fateman. Polynomial multiplication, powers, and asymptotic analysis: some comments. SIAM J. Comput. **3**, 3 (1974), pp. 196–213.
3. H. Fettis. Algorithm 158. Communications of the ACM, 6 (1963), pp. 104.
4. M. Gentleman. Optimal multiplication chains for computing a power of a symbolic polynomial. Math Comp. **26**, 120 (1972), pp. 935–939.
5. L. Heindel. Computation of powers of multivariate polynomials over the integers. J. Comput. Syst. Sci. **6**, 1 (1972), pp. 1–8.
6. E. Horowitz, S. Sahni. The computation of powers of symbolic polynomials. SIAM J. Comput. **4**, 2 (1975), pp. 201–208.
7. S.C. Johnson. Sparse polynomial arithmetic. *ACM SIGSAM Bulletin*, **8** (3) 63–71, 1974.
8. D. Knuth. The Art of Computer Programming, Volume 2: *Seminumerical Algorithms.* Addison-Wesley (1998).
9. R. Moenck. Another Polynomial Homomorphism. *Acta Informatica*, **6**, 153–169, 1976.
10. C. Ponder. Parallel multiplication and powering of polynomials. *J. Symbolic. Comp.,* **11** (4), 307–320, 1991.
11. D. Probst, V. Alagar. A Family of Algorithms for Powering Sparse Polynomials. SIAM J. Comput. **8**, 4 (1979), pp. 626–644.
12. M. Monagan, R. Pearce. Parallel Sparse Polynomial Multiplication Using Heaps. *Proc. of ISSAC 2009*, ACM Press, 295–315.
13. M. Monagan, R. Pearce. Polynomial Division Using Dynamic Arrays, Heaps, and Packed Exponent Vectors. *Proc. of CASC 2007*, Springer LNCS 4770, 295–315.
14. M. Monagan, R. Pearce. Sparse Polynomial Division Using a Heap. *J. Symbolic. Comp.,* **46** (7), 807–922, 2011.
15. M. Monagan, R. Pearce. Parallel Sparse Polynomial Division Using Heaps. *Proc. of PASCO 2010*, ACM Press, 105–111, 2010.
16. W. Rowan. Efficient Polynomial Substitutions of a Sparse Argument. *ACM Sigsam Bulletin*, **15** (3), 17–23, 1981.
17. D. Zeilberger. The J.C.P. Miller recurrence for exponentiating a polynomial, and its q-analog. *J. Difference Eqns and Appls*, **1** (1), 57–60, 1995. `http://www.math.rutgers.edu/~zeilberg/mamarim/mamarimPDF/power.pdf`