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, Maple 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 i^{th} 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.

The methods in the literature are as follows.

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

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

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 $g = f^k$.
$g :=$ an array with $kd + 1$ elements indexed from zero
$g_{kd} := f_d^k$
for i from $kd - 1$ to 0 by -1 do
  $e := kd - i$
  $c := \sum_{j=1}^{\min(d,e)} ((k+1)j - e) \cdot f_{d-j} \cdot g_{i+j}$
  $g_i := c/(e \cdot f_d)$
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 non-zero products. The heap is used to merge the set of all pairwise term products $f_i \times g_j$ in descending order. We exploit the fact that the term $f_i \times g_j$ is strictly greater than $f_i \times g_{j+1}$ and $f_{i+1} \times g_j$ to reduce the size of the heap. In lines 12 and 13, we avoid having multiple f_i in the heap with the same g_j. 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. For dense polynomials, we also use an optimization called *chaining* to combine products with equal monomials, see [13, 14].

2.1 Complexity and Optimizations

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. It stores g and uses $O(\#f)$ additional memory.*

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 $2 \leq i \leq \#f$ and $1 \leq j \leq \#g$ 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. The objects stored other than g are the heap H and set Q which have at most $\#f$ entries. □

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 do not occur in the 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 coefficient arithmetic in *Sparse SUMS*. And, because we divide by the exponents, we cannot 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 inefficient. When sparse polynomials, e.g. those arising from a Kronecker substitution, are raised to a low power, typically $\#f^k \gg \#f^{(k-1)}$. The cost of *RMUL* will be mostly in the final step which does $\#f \cdot \#f^{(k-1)}$ multiplications. But *Sparse SUMS* does $O(\#f \times \#f^k)$ coefficient operations, which could be far more in total.

We note that *Sparse SUMS* could construct f^{k+1} almost for free, because it already multiplies every term of f^k by every term of f except for f_1. To exploit this we created a variant that we call *FPS*. It uses the *Sparse SUMS* algorithm to compute f^{k-1} and outputs f^k as a side effect.

We present *FPS* at the end of this section by adding lines to our description of *Sparse SUMS*. To reduce the number of coefficient operations, lines 9 and 11 should reuse $cof(f_i) \cdot cof(g_j)$, and lines 17 and 18 should update C and S with $C := C/(\deg(g_1) - M);\ S := S + C;\ C := C/cof(f_1)$.

Table 1 counts coefficient multiplications to compare the cost of the sparse 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* adds a factor of $(2t - 1)k/(k + t - 1)$. Those methods also do many divisions in the sparse case, however their cost does 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 i^{th} term by at least $x^{(k+1)^{i-2}}$ for $i > 2$, so the degree of f^k is $d = k\,(k + 1)^{t-2}$.

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	$\frac{(k+t-1)!}{k!(t-1)!} + \cdots$	$k^2(k-1)(t-2)^2/24 + \cdots \in O(k^3t^2)$
SUMS	$\frac{(2t-1)(k+t-1)!}{k!(t-1)!}$	$(2t-1)((t-1)k+1) \in O(kt^2)$
FPS	$\frac{(2t-1)(k+t-2)!}{(k-1)!(t-1)!}$	$(2t-1)((t-1)(k-1)+1) \in O(kt^2)$

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* compute the result in 1.64×10^5 and 1.55×10^5 operations. In the dense case, *SUMS* and *FPS* are $O(kt^2)$ and the other sparse algorithms are $O(k^2t^2)$. The FFT is $O(d\log d)$ where $d = ((t-1)k+1)$ is now the size of the result, however, *SUMS* can still win if $\log d > 2t$, that is, *SUMS* is the best method for raising small dense polynomials to high powers.

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 $h = f^k$.

$H :=$ an empty heap ordered by degree with maximum element H_1
$g := f_1^{k-1}$; $h := f_1^k$
insert $f_2 \times g_1 = (2, 1, \deg(f_2) + \deg(g_1))$ into H
while $|H| > 0$ do
  $M := \deg(H_1)$; $C := 0$; $S := 0$; $Q := \{\}$;
  while $|H| > 0$ and $\deg(H_1) = M$ do
    $(i, j, M) := extract\_max(H)$
    $(\alpha, \beta) := (degree(f_i), degree(g_j))$
    $S := S + cof(f_i) \cdot cof(g_j)$
    if $M \geq \deg(f_1)$ and $\beta \neq (k-1)\alpha$
      then $C := C + (\beta - (k-1)\alpha) \cdot cof(f_i) \cdot cof(g_j)$
    $Q := Q \cup \{(i, j)\}$
  for all $(i, j) \in Q$ do
    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
    if $i < \#f$ and $f_{i+1} \times g$ not in H then insert $f_{i+1} \times g_j$ into H
  if $C \neq 0$ then
    $C := C/((\deg(g_1) - M) \cdot cof(f_1))$
    $S := S + C \cdot cof(f_1)$
    $g := g + C\, x^{M - \deg(f_1)}$
    if $f_2 \times g$ has no term in H then insert $f_2 \times g_{\#g}$ into H
  if $S \neq 0$ then $h := h + S\, x^M$
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.

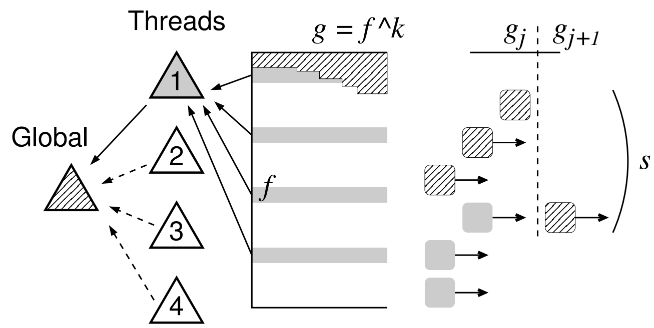

Figure 1 shows features common to all our parallel algorithms. The work of merging products $f_i \times g_j$ is divided into strips along the terms of f, so threads are given subsets of f to multiply by g. A global function combines their results and computes new terms of g. This function is protected 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 for completely sparse (t terms)^k.

input		result			C code				Magma		Singular
t	k	*terms*	*degree*	*bits*	*SUMS*	*FPS*	*RMUL*	*BINA*	*FFT*	*RSQR*	*RMUL*
3	100	5151	10100	152	0.001	0.001	0.026	0.001	0.01	0.25	0.05
3	250	31626	62750	388	0.007	0.013	0.484	0.011	0.45	12.84	1.04
3	500	125751	250500	784	0.035	0.069	4.560	0.055	3.48	278.13	12.75
3	1000	501501	1001000	1575	0.208	0.414	45.664	0.333	31.38	–	125.29
3	2500	3128751	6252500	3951	2.328	4.770	–	5.667	(*)	–	–
4	50	23426	130050	92	0.005	0.007	0.033	0.005	0.12	1.34	0.18
4	100	176851	1020100	191	0.040	0.073	0.763	0.055	3.10	98.71	2.49
4	200	1373701	8080200	389	0.373	0.714	13.151	0.521	74.36	–	44.61
4	400	10827401	64320400	788	3.636	7.405	247.743	5.144	–	–	889.79
6	20	53130	$3.89 \cdot 10^6$	42	0.008	0.008	0.021	0.006	3.25	0.77	0.07
6	30	324632	$2.77 \cdot 10^7$	67	0.056	0.057	0.173	0.039	63.27	26.00	1.17
6	40	1221759	$1.13 \cdot 10^8$	91	0.332	0.531	1.471	0.222	–	460.42	6.67
6	50	3478761	$3.38 \cdot 10^8$	117	1.000	1.682	6.547	0.838	–	–	26.89
6	70	17259390	$1.78 \cdot 10^9$	167	5.123	9.256	49.476	5.029	–	–	176.80
8	15	170544	$2.51 \cdot 10^8$	34	0.031	0.027	0.052	0.023	(*)	0.95	0.10
8	20	888030	$1.71 \cdot 10^9$	47	0.179	0.162	0.337	0.117	–	36.20	1.84
8	25	3365856	$7.72 \cdot 10^9$	62	0.677	0.649	1.504	0.452	–	284.64	10.70
8	30	10295472	$2.66 \cdot 10^{10}$	76	2.838	3.135	6.143	1.546	–	–	42.92
8	35	26978328	$7.62 \cdot 10^{10}$	90	9.042	13.828	28.342	5.927	–	–	148.97
12	10	352716	$2.59 \cdot 10^{11}$	22	0.088	0.055	0.074	0.050	–	1.61	0.18
12	12	1352078	$1.65 \cdot 10^{12}$	29	0.364	0.231	0.330	0.199	–	11.84	0.89
12	14	4457400	$8.07 \cdot 10^{12}$	35	1.222	0.864	1.220	0.672	–	78.81	4.06
12	16	13037895	$3.22 \cdot 10^{13}$	41	3.538	2.631	3.970	1.982	–	500.20	21.99
12	18	34597290	$1.10 \cdot 10^{14}$	47	9.339	7.166	11.468	5.402	–	–	(*)
12	20	84672315	$2.15 \cdot 10^{14}$	54	22.537	18.071	29.922	13.360	–	–	–

– 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 timings are for version 2.17. Singular timings are for version 3.10. Timings for *SUMS*, *FPS*, *RMUL*, and *BINA* are real times from our C library. For Magma and Singular we report CPU timings, which we found to be less precise.

3.1 Sparse Problems

To create polynomials with t terms whose powers up to k are completely sparse, we used 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 *SUMS* is not as disadvantageous as it may first appear.

Table 2 compares *SUMS*, *FPS*, *RMUL* and *BINA*. The polynomials are too short to run our parallel algorithms. For Magma we give two times; FFT is the *RSQR* algorithm with Schönhage-Strassen multiplication. We also tried writing the problem as multivariate, which uses *RSQR* and sparse arithmetic. Singular uses sparse arithmetic and *RMUL* which is a sensible choice.

The timings show that *SUMS* is consistently faster than *RMUL*, and is the fastest method for higher powers in fewer variables. The *FPS* method becomes slower relative to *SUMS* as k increases but faster as t increases. *BINA* is highly competitive in all cases, and is the fastest method tested for 6 or more terms.

3.2 Dense Problems

Table 3 shows timings for expanding powers of the polynomial

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

Dense problems are a strong case for *SUMS*. *RMUL* and *BINA* are competitive only for low powers of large polynomials, where the FFT is the fastest method. This implies *SUMS* is the best sparse method to complement the FFT. Higher powers benefit *SUMS* versus the FFT. For 500 terms, *SUMS* goes from 21 times slower at $k = 10$ down to 1.5 times slower at $k = 320$, breaking even at $k = 640$. Our parallel speedup appears to be limited to 3.8. The timings for *FPS* do not fit in the table, but they are slower than *SUMS* by a factor of 3. We think this ratio will improve with optimization.

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

t	k	*SUMS*	*RMUL*	*BINA*	*FFT*	t	k	*SUMS*	(4 cores)	*RMUL*	*FFT*
10	200	0.001	0.085	0.098	0.006	500	10	0.084	0.026	0.151	0.004
10	500	0.005	0.752	1.078	0.095	500	20	0.198	0.058	1.343	0.014
10	1000	0.015	4.474	8.178	0.501	500	40	0.476	0.131	6.944	0.057
10	1500	0.032	13.386	29.630	0.510	500	80	1.200	0.322	34.933	0.247
10	2000	0.055	29.808	–	2.640	500	160	3.351	0.921	192.162	1.352
10	2500	0.082	55.433	–	2.670	500	320	10.616	2.808	–	6.890
100	50	0.023	0.415	0.428	0.017	1000	3	0.045	0.015	0.034	0.001
100	100	0.057	2.087	2.165	0.056	1000	5	0.078	0.026	0.115	0.003
100	200	0.159	11.091	11.728	0.262	1000	10	0.361	0.102	0.797	0.013
100	400	0.497	66.643	71.487	1.360	1000	20	0.824	0.228	5.714	0.030
100	800	1.730	446.477	–	6.990	1000	40	1.951	0.525	29.393	0.130
100	1600	6.087	–	–	36.310	1000	80	5.035	1.325	149.326	0.570

Table 4 reports the time to power two dense multivariate polynomials. The data shows that conventional sparse methods (*RMUL* and *BINA*) beat the FFT as the number of variables increases. Because it has better complexity on dense problems, *SUMS* has a much easier time beating the FFT. It gains more as the power k or the number of variables is increased.

Table 4. Timings for dense multivariate f^k.

$f=(1+x+y)^{15}$ $t=136$								Magma	Singular
k	$\#g$	*SUMS*	4 cores	*FPS*	*RMUL*	4 cores	*BINA*	*FFT*	*RMUL*
20	45451	0.536	0.149	0.685	1.514	0.429	1.553	0.49	12.33
40	180901	3.157	0.846	4.181	15.833	4.406	16.375	5.49	134.59
60	406351	9.263	2.478	12.552	65.276	17.927	66.790	27.27	522.59
80	721801	20.439	5.402	28.110	182.717	49.830	187.178	56.42	–
120	1622701	64.117	16.618	88.688	–	–	–	325.60	–

$f=(1+w+x+y+z)^4$ $t=70$								Magma	Singular
k	$\#g$	*SUMS*	2 cores	*FPS*	*RMUL*	2 cores	*BINA*	*FFT*	*RMUL*
4	4845	0.005	0.005	0.003	0.003	0.003	0.003	0.30	0.01
8	58905	0.068	0.062	0.048	0.071	0.047	0.072	1.24	1.01
12	270725	0.711	0.440	1.021	0.955	0.589	0.995	10.84	10.40
16	814385	2.311	1.297	3.784	5.238	3.120	5.443	65.50	46.49
20	1929501	5.852	4.755	10.337	17.164	10.065	17.790	218.14	166.02
24	3921225	12.313	11.350	22.643	44.008	25.513	45.489	391.42	394.08
28	7160245	23.430	22.754	45.458	97.179	56.745	100.277	(*)	–

The only case where *SUMS* loses to *RMUL* or *BINA* is $k=4$ in the second problem. In that case, and also for $k=8$, the *FPS* algorithm does much better. The parallel speedup for *SUMS* is good on the first problem but it deteriorates on the second problem as k increases. We suspect the routine is struggling with data dependencies because parallel division of f^k by f shows the same issue.

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 it in reasonable time. This can now be done quickly. Table 5 shows that *SUMS* is the fastest 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 will use sparse *RSQR*, which takes 134.49 seconds for $k=40$.

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

	result	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.

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 $(1 + 3x + 2x^2)^{3000}$."

Zeilberger coded dense *SUMS* in Maple and noted that it was theoretically faster than the FFT, although his analysis does not account for the coefficients which exceed 2300 decimal digits. At the time Maple was using *BINA*, which is a poor choice on this problem as it needs over 2 GB of memory to store all the expanded powers of $(3x + 1)^i$ for i up to 3000.

Table 6 shows that *SUMS* is by far the fastest method on this example. The digits column shows the length in decimal digits of the largest coefficient in the result. By default, Maple 16 uses *RSQR* and performs univariate multiplication by evaluating at a suitable power of 2 and leveraging the FFT from fast integer multiplication. This takes 1 second on our Intel Core i7 2.66 GHz machine. But *SUMS* takes under 9 milliseconds! It does fewer than $2t^2k = 2 \cdot 9 \cdot 3000 = 54000$ coefficient multiplications; and because the coefficients of f are small, at most half of those are multiprecision.

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

		C code			Magma	Maple 16	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.

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.

One improvement that we can make is to generate the terms of the output $g = g_1 + g_2 + \cdots + g_m$ from both directions in parallel and meet in the middle.

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*.

Acknowledgment

We thank the referees for their suggestions which have improved this paper.

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`