Parallel Sparse Polynomial Powering Using Heaps

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 new algorithm for expanding powers of sparse distributed polynomials on a multicore processor. The new method has better complexity and lower space requirements than previous algorithms. It is designed to run in the cache and achieve superlinear speedup. On a series of benchmark problems, we compare our new algorithm implemented in C to previous methods and to the routines of other computer algebra systems.

Categories and Subject Descriptors: I.1.2 [Symbolic and Algebraic Manipulation]: Algebraic Algorithms

General Terms: Algorithms, Design, Performance

Keywords: Parallel, Sparse, Polynomial, Power, Heaps

1. INTRODUCTION

Expanding powers of sparse polynomials is an elementary operation in computer algebra systems. Despite receiving a great deal of attention in the 1970's, a fragmented situation exists today where the fastest algorithms all make time and memory tradeoffs that are undesirable in some cases. Thus, programmers of computer algebra systems must implement multiple routines and select carefully 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 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 $f = x_1 + x_2 + \cdots + x_t$. Raising f to the power k produces $\binom{k+t-1}{k}$ terms, the most possible. In the dense case, the terms of f combine as much as possible, e.g. as in $f = x + x^2 + \cdots + x^t$. If there are no cancellations, f^k will have $k(t-1)+1$ terms.

*We gratefully acknowledge the support of the MITACS NCE of Canada and NSERC of Canada

We wanted a sparse algorithm that performs well enough in both cases that it can be called without reservation. The most prominent methods in the literature the following:

RMUL computes $f^i = f \cdot f^{i-1}$ for $i = 2 \ldots k$. The memory taken by f^{i-2} is reused to hold f^i in our implementation.

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

It was first pointed out by Gentleman [4] and Heindel [5] that *RSQR* is vastly slower than *RMUL* in the sparse case. *RMUL* can conserve memory by using an online algorithm to multiply $f \cdot f^{i-1}$ using $O(\#f)$ temporary storage [8, 14]. The total memory can be kept to less than twice the result.

BINA picks a term $f_1 \in f$ and expands $g = (f_1 + 1)^k$ with the binomial theorem. It computes $(f - f_1)^i$ for $i = 2 \ldots k$ by multiplication and merges $f^k = \sum_{i=0}^{k} g_i \cdot (f - f_1)^i$. Our routine avoids unbalanced merging by storing the products and doing a simultaneous n-ary merge.

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

The binomial algorithms were introduced by Fateman [1], who shows that *BINB* is nearly optimal in the sparse case. Alagar and Probst [12] improve on it by using recursion, and Rowan [17] expands the set of powers $\{g^i\}$ more efficiently, both in the sparse case only. In the dense case, Fateman [2] shows that *BINA* is much faster than *BINB*.

MNE generates all combinations of terms and multinomial coefficients directly. See [6]. It sorts to combine like terms. This quickly becomes infeasible in the dense case.

FFT performs multipoint evaluations and interpolations at roots of unity modulo primes. Chinese remaindering is used to recover the result.

As noted by Ponder in [11], the FFT is often competitive in practice because powers of a polynomial tend to "fill-in". For multivariate polynomials, one should use the Kronecker substitution as suggested by Moenck [10]. Finally, consider this classical method for power series due to Euler [2, 3, 9].

SUMS is a dense method. Let $f = \sum_{i=0}^{d} f_i x^i$. To compute $g = f^k = \sum_{i=0}^{kd} g_i x^i$ it powers $g_0 = f_0^k$ and uses 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 complexity of these methods in the sparse and dense cases is shown in Table 1. For the sparse case, we also note an important theorem from Fateman [1].

THEOREM 1. *No algorithm can compute $g = f^k$ in fewer than $\#g - \#f$ coefficient multiplications in the sparse case.*

Table 1: Cost to power $(t$ terms$)^k$.

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

Table 1 counts coefficient multiplications and divisions to estimate the cost of each method. In the sparse case, *BINB* approaches the size of the result, *RMUL* adds a factor of k, and *BINA* adds a factor of $kt/(k+t-1)$. Sparse *SUMS* is the method of this paper which adds a factor of $2t-1$.

In the dense case, *SUMS* is the best method, *RMUL* and *BINA* add factors of k, and *BINB* adds a factor of kt. Our impression is that improvements to the sparse case damage performance in the dense case, which may be a bad tradeoff in practice. Most problems become dense as k increases.

The paper is organized as follows. Section 2 develops the algorithm and presents a complexity analysis. We find that sparse *SUMS* is highly competitive in both time and space for the sparse and dense cases. Section 3 describes how we parallelized the algorithm for modern multicore processors. In Section 4 we present benchmarks of our implementation.

2. SPARSE SUMS

Two features of *SUMS* suggest a sparse algorithm: it has an outer loop constructing terms of the result in order, and its inner loop computes terms by adding pairwise products. We can safely skip over products where f_j or g_{i-j} are zero. The formula then describes a multiplication of $g = f^k$ by f where pairwise products are scaled and merged to compute new terms of g.

Our first task is to modify *SUMS* to compute new terms in descending order, dividing by the leading coefficient of f instead of the constant term f_0. This simplifies the method in the sparse case and also handles the problem of $f_0 = 0$.

Algorithm: SUMS (descending order).
Input: dense polynomial $f = f_0 + f_1x + \cdots + f_dx^d$, $f_d \neq 0$, stored as an array $[f_0, f_1, \ldots, f_d]$ indexed from zero.
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

In the algorithm above, 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 the 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 $((k+1)j-e) = \beta - k\alpha$.

Algorithm: Sparse SUMS.
Input: sparse polynomial $f = f_1 + f_2 + \cdots + f_t$, $f_i > f_{i+1}$.
monomial ordering $<$.
positive integer k.
Output: sparse polynomial $g = f^k$.
$H :=$ an empty heap ordered by $<$ with max element H_1
$g := f_1^k$
insert $f_2 \times g_1 = (2, 1, mon(f_2)\cdot mon(g_1))$ into H
while $|H| > 0$ and $\deg(H_1) \geq \deg(f)$ do
$M := mon(H_1)$; $C := 0$; $Q := \{\}$;
while $|H| > 0$ and $mon(H_1) = M$ do
$(i, j, M) := extract\_max(H)$
$(\alpha, \beta) := (degree(f_i), degree(g_j))$
$C := C + (\beta - k\alpha)\cdot cof(f_i)\cdot cof(g_j)$
$Q := Q \cup \{(i, j)\}$
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
insert $f_{i+1} \times g_j$ into H
if $C \neq 0$
$C := C/((degree(g_1) - degree(M))\cdot cof(f_1))$
$M := M/mon(f_1)$
$g = g + C\cdot M$
if $f_2 \times g$ has no term in H
insert $f_2 \times g_{\#g}$ into H
return g

The sparse version of *SUMS* is presented above. It uses a heap of pointers into f and g to combine only the products $f_i \times g_j$ with f_i and g_j non-zero. The approach is described in further detail in [15]. The idea is to use a heap to merge the set of all pairwise products $f_i \times g_j$ in descending order. Properties of the monomial ordering are used to reduce the number of products compared in the heap at any one time. In particular, we insert $f_i \times g_{j+1}$ after merging $f_i \times g_j$, and only if $f_{i-1} \times g_{j+1}$ has already been merged.

In computer memory, the heap is an array of size $O(\#f)$ with pointers into a second array with the products $f_i \times g_j$. For each $f_i \in f$, we store 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 of whether $f_{i-1} \times g_j$ has been merged easy. We simply test if the pointer for f_{i-1} has advanced further than g_j.

In the dense case, the chaining optimization we described in [15] is used to amalgamate terms with equal monomials. This reduces the cost of heap operations from $O(\log \#f)$ to $O(1)$ so that the overhead of the heap is negligible. A proof is given in [14]. For multivariate polynomials we employ the Kronecker substitution to handle the problem as univariate. In particular, we use a bit-packing scheme described in [15] to store all of the exponents in one machine word.

THEOREM 2. *The sparse sums algorithm expands $g = f^k$ using $(2\,\#f - 1)\,\#g + 2\log k$ coefficient multiplications and $\#g$ divisions and $O(\#f\,\#g\log\#f)$ monomial comparisons. It uses $O(\#f + \#g)$ memory.*

PROOF. For $g_1 = f_1^k$ binary powering 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 $1 \leq j \leq \#g$ using the heap, performing two coefficient multiplications and $\log \#f$ comparisons each. This does not count the multiplication in $\beta - k\alpha$. For each term of the result g, we perform one multiplication and one division to compute its coefficient. The only objects stored are the heap of size $\#f$ and the result $\#g$. The terms of g are constructed to satisfy $f\cdot g' = k\cdot g\cdot f'$ so $g = f^k$. This may be verified by induction on the dense version. □

3. PARALLELIZATION

Our design for a parallel version of this algorithm follows the approach we used for sparse polynomial division in [16]. Both algorithms share exactly the same problem, which is a tight data-dependency among the terms in the result. Each new term of g may depend on any subset of previous terms with no predictable pattern since the problem is sparse. To create parallelism we split the computation into interacting pieces and exploit the problem's structure to hide latencies in communication. We avoid synchronization at all costs.

Figure 1: Components of the parallel algorithm.

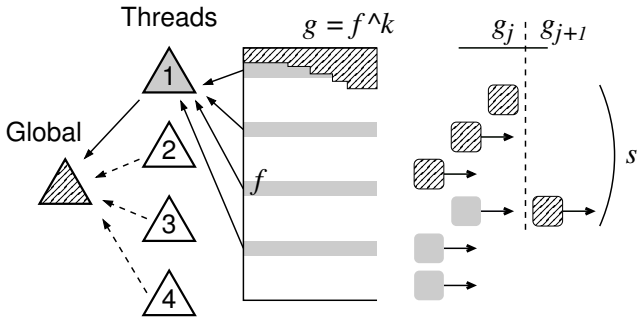

Figure 1 shows numerous design features common to our parallel algorithms for sparse polynomial multiplication and and division in [13, 16]. The work of merging products $f_i g_j$ is divided into strips along the terms of f, so processors are given subsets of f to multiply by g. The working storage of the heap is similarly divided, which allows the algorithm to achieve superlinear speedup. There is a global function that combines these results and computes new terms of g. This function is protected by a lock and can be called by any of the threads. That allows the threads to collectively balance their loads by performing different amounts of global work.

Another feature is borrowed from [16] and 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 communication delay for $f_2 \times g_j$. Recall that this term must be compared to all others immediately, as it could be merged next in the computation.

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 $f_i \times g_{j+1}$ into their heap, only to find that g_{j+1} does not exist. This can occur for a number of reasons. Perhaps the term has simply not been computed yet. But a more sinister reason is that $f_i \times g_j$ may not have produced a new term of g, so the global function now needs $f_{i+1} \times g_j$ to make 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 nt contains the number of terms in the result we have computed so far, while rs contains the number of rows stolen by the global function. To ensure correctness, the threads read rs before nt and that order must be enforced using memory barriers. Likewise, the global function must always update nt before incrementing rs. Updating nt means that a new term of g was computed, and along with its monomial and coefficient the global function stores a copy of the current value of rs. This ensures that the threads know exactly which products $f_i \times g_j$ were stolen and computed by the global function.

Subroutine: Global Function.
Input: updates the global state of the algorithm.
Output: constructs new terms of $g = f^k$ when possible.
Globals: global heap G, sets B and Q, polynomials f, g.
rows stolen rs, number of terms computed nt
integer power k

```
// B is the set of buffers with terms from threads
for all buffers b in B do
  if buffer b is not empty then
    extract a term (c, m) from buffer b
    insert [b, c, m] into global heap G for merging
  else if buffer b has been closed by its thread
    discard buffer b from B (thread is done)
  else
    rs := rs + 1 (steal a new row)
    return (to the calling thread)
// now we can merge terms
m := mon(G1); c := 0; Q := {};
while |G| > 0 and deg(G1)
  (i, j, m) := extract_max(G)
  if i points to a buffer
    c := c + j (scaled by thread)
  else
    (α, β) := (degree(fi), degree(gj))
    c := c + (β − kα) · cof(fi) · cof(gj)
    Q := Q ∪ {(i, j)}
for all (i, j) ∈ Q do
  if j < nt and (i = 1 or f_{i−1} × g_{j+1} was merged)
    insert fi × g_{j+1} into G
  if i < #f and i < rs(gj) and f_{i+1} × g not in G
    insert f_{i+1} × gj into G (insert stolen row)
if c ≠ 0 then
  c := c/((degree(g1) − degree(m)) · cof(f1))
  m := m/mon(f1)
  g := g + [c · m, rs]
  nt := nt + 1
  if f2 × g has no term in G
    insert f2 × g_nt into G
return (to the calling thread)
```

Subroutine: Thread Function.
Input: array of assigned indices t in f, buffer b.
Output: writes terms to the buffer b when possible.
Globals: rows stolen rs, number of terms computed nt
lock L, polynomials f and g, integer power k

```
insert f_{t1} × g1 into H
while |H| > 0 and deg(H1) ≥ deg(f) do
  m := mon(H1); c := 0; Q := {};
  while |H| > 0 and mon(H1) = m do
    (i, j, m) := extract_max(H)
    (α, β) := (degree(fi), degree(gj))
    c := c + (β − kα) · cof(fi) · cof(gj)
    Q := Q ∪ {(i, j)}
  for all (i, j) ∈ Q do
    γ := minimum value in t larger than i
    if γ exists and f_γ × g not in H
      insert f_γ × gj into H
    retry:
    RS := rs; NT := nt;
    if j < NT and rs(gj) < i
      insert fi × g_{j+1} into H
    else if j < NT or j ≤ RS // discard stolen fi × gj
      γ := minimum value in t larger than i
      if γ exists and f_γ × g not in H
      i := γ; goto retry:
    else if trylock(L)
      global_function() // compute terms of g
      goto retry:
    else
      spinwait() // thread is blocked
      goto retry:
  close buffer b
```

4. BENCHMARKS

Our benchmarks were performed on an Intel Core i7 920 2.66GHz running 64-bit Linux. A salient feature of this cpu is the shared L3 cache that allows communication between the cores at high throughput and low latency. Our software (sdmp) is designed to perform well on any cpu, but for fine grained data-dependencies a shared cache is important.

4.1 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

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

Observe that one 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. We compare the times for our sequential algorithm to *RMUL* and *BINA* which we previously implemented in C.

Table 2: Time for completely sparse $(t$ terms$)^k$.

t	k	$\#g$	*SUMS*	*BINA*	*RMUL*
3	100	5151	0.001 s	0.001 s	0.030 s
3	250	31626	0.010 s	0.010 s	0.480 s
3	500	125751	0.050 s	0.050 s	4.570 s
3	1000	501501	0.320 s	0.290 s	45.630 s
3	2500	3128751	4.130 s	5.260 s	–
4	50	23426	0.005 s	0.005 s	0.030 s
4	100	176851	0.060 s	0.060 s	0.760 s
4	200	1373701	0.480 s	0.480 s	13.308 s
4	400	10827401	5.180 s	5.160 s	252.230 s
5	40	135751	0.030 s	0.020 s	0.130 s
5	60	635376	0.180 s	0.130 s	0.580 s
5	80	1929501	0.580 s	0.460 s	6.460 s
5	100	4598126	1.530 s	1.280 s	19.950 s
5	120	9381251	3.240 s	2.690 s	50.170 s
5	140	17178876	6.320 s	5.190 s	110.340 s
6	20	53130	0.010 s	0.010 s	0.030 s
6	30	324632	0.060 s	0.040 s	0.190 s
6	40	1221759	0.370 s	0.220 s	1.500 s
6	50	3478761	1.150 s	0.810 s	6.670 s
6	60	8259888	2.770 s	2.170 s	20.170 s
6	70	17259390	6.050 s	4.840 s	51.140 s
8	15	170544	0.020 s	0.020 s	0.050 s
8	20	888030	0.190 s	0.130 s	0.380 s
8	25	3365856	0.730 s	0.490 s	1.670 s
8	30	10295472	3.030 s	1.620 s	6.670 s
8	35	26978328	10.030 s	6.090 s	29.950 s
12	10	352716	0.060 s	0.050 s	0.070 s
12	12	1352078	0.280 s	0.210 s	0.360 s
12	14	4457400	1.060 s	0.760 s	1.340 s
12	16	13037895	3.170 s	2.250 s	4.330 s
12	18	34597290	8.570 s	6.260 s	12.680 s
20	8	2220075	0.490 s	0.330 s	0.390 s

We see that sparse sums performs about as well as *BINA* on sparse problems, while using far less intermediate space. This is because for *BINA* to run as fast as it does here, we must store the polynomials $(f - f_1)^i$ for $1 \le i \le k$ and use a simultaneous n-ary merge to compute the result. Nothing is combined on these sparse problems, but the routine must detect like terms when they do exist. *RMUL* and *BINA* use intermediate space that is $O(\#g)$ while *SUMS* is only $O(t)$. All of the algorithms perform well enough as the number of terms increases.

4.2 Dense Problems

Dense problems are the best case for *SUMS*, but at lower powers *RMUL* and *BINA* provide stiff competition. *SUMS* probably should not be used to square or cube polynomials but for higher powers it is superior. This benchmark shows the effect of parallelization as well, since *RMUL* and *BINA* use our parallel multiplication routine [13]. The thresholds for starting $\{2, 3, 4\}$ threads are $\#f = \{64, 216, 512\}$ terms. The base time is our sequential implementation followed by parallel speedup with the highest number of threads. When this measurement is insufficiently precise we denote that by a star. The polynomial $f = 1 + x + x^2 + \cdots + x^{t-1}$.

Table 3: Time for completely dense $(t$ terms$)^k$.

t	k	*SUMS*		*RMUL*		*BINA*	
10	200	0.000 s	–	0.085 s	–	0.095 s	–
10	500	0.005 s	–	0.760 s	–	1.035 s	–
10	1000	0.020 s	–	4.450 s	–	7.930 s	–
10	1500	0.040 s	–	13.370 s	–	28.950 s	–
10	2000	0.070 s	–	29.840 s	–	–	–
100	50	0.020 s	*	0.420 s	1.8x	0.420 s	1.8x
100	100	0.055 s	*	2.090 s	1.8x	2.110 s	1.8x
100	200	0.155 s	*	11.090 s	1.8x	11.425 s	1.8x
100	400	0.490 s	1.7x	65.980 s	1.9x	69.565 s	1.8x
100	800	1.700 s	1.7x	439.380 s	1.9x	–	–
500	10	0.085 s	*	0.150 s	*	0.150 s	*
500	20	0.185 s	*	1.330 s	2.7x	1.330 s	2.6x
500	40	0.440 s	2.5x	6.955 s	2.7x	6.955 s	2.7x
500	80	1.130 s	2.5x	35.075 s	2.8x	35.120 s	2.8x
500	160	3.235 s	2.6x	190.985 s	2.8x	70.030 s	2.8x
1000	3	0.035 s	*	0.035 s	*	0.035 s	*
1000	5	0.065 s	*	0.115 s	*	0.115 s	*
1000	10	0.345 s	3.5x	0.785 s	3.7x	0.790 s	3.6x
1000	20	0.765 s	3.6x	5.735 s	3.8x	5.725 s	3.7x
1000	40	1.840 s	3.7x	29.250 s	3.9x	29.190 s	3.8x
1000	80	4.790 s	3.7x	148.835 s	4.0x	148.450 s	3.8x

Obviously with dense polynomials one should switch to a dense algorithm like the FFT as t increases. It is hard to make a case for parallelizing *SUMS* based on the data here. However multivariate polynomials are raised to moderately high powers by various routines in Maple and in those cases a parallel sparse powering algorithm appears warranted. We are in the process of collecting good examples. The parallel speedup for *RMUL* is hindered by the restarting of threads. The parallel speedup for *BINA* is worse because the n-ary merge at the end is sequential. The speedup for *SUMS* was lousy but our parallel code is only preliminary.

4.3 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 following polynomial to high powers which no computer algebra system could do in a reasonable amount of time. We thought it should be possible, and now it might be, if only we could store the result.

$$\begin{aligned} f = {} & xy^3z^2 + x^2y^2z + xy^3z + xy^2z^2 + y^3z^2 + y^3z \\ & + 2y^2z^2 + 2xyz + y^2z + yz^2 + y^2 + 2yz + z \end{aligned} \tag{1}$$

Table 4: Time to power f^k.

k	$\#g$	*SUMS*	*RMUL*	*BINA*
40	243581	0.150 s	0.980 s	0.840 s
70	1284816	0.970 s	11.080 s	9.330 s
100	3721951	3.050 s	49.690 s	42.930 s
150	12499176	11.750 s	276.320 s	–
200	29553901	30.870 s	–	–

5. CONCLUSION

We have presented what may be the best general purpose algorithm for powering sparse polynomials, and shown how to parallelize it despite its inherently sequential nature. An obvious question is whether this was worth doing. We think it was, because when basic algorithms in computer algebra are suboptimal it reflects poorly on the field. This problem lacked a satisfactory algorithm for many years so it pleases us to present a solution even if major applications are rare.

6. 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] E. Horowitz. The Efficient Calculation of Powers of Polynomials. J. of Comp. Sys. Sci. **7**, 5 (1973), pp. 469–480.

[8] S.C. Johnson. Sparse polynomial arithmetic. *ACM SIGSAM Bulletin*, **8** (3) 63–71, 1974.

[9] D. Knuth. The Art of Computer Programming, Volume 2: *Seminumerical Algorithms.* Additson-Wesley (1998).

[10] R. Moenck. Another Polynomial Homomorphism. *Acta Informatica*, **6**, 153–169, 1976.

[11] C. Ponder. Parallel multiplication and powering of polynomials. *J. Symbolic. Comp.,* **11** (4), 307–320, 1991.

[12] D. Probst, V. Alagar. A Family of Algorithms for Powering Sparse Polynomials. SIAM J. Comput. **8**, 4 (1979), pp. 626–644.

[13] M. Monagan, R. Pearce. Parallel Sparse Polynomial Multiplication Using Heaps. *Proc. of ISSAC 2009*, ACM Press, 295–315.

[14] M. Monagan, R. Pearce. Polynomial Division Using Dynamic Arrays, Heaps, and Packed Exponent Vectors. *Proc. of CASC 2007*, Springer LNCS 4770, 295–315.

[15] M. Monagan, R. Pearce. Sparse Polynomial Division Using a Heap. *J. Symbolic. Comp.,* **46** (7), 807–922, 2011.

[16] M. Monagan, R. Pearce. Parallel Sparse Polynomial Division Using Heaps. *Proc. of PASCO 2010*, ACM Press, 105–111, 2010.

[17] W. Rowan. Efficient Polynomial Substitutions of a Sparse Argument. *ACM Sigsam Bulletin*, **15** (3), 17–23, 1981.