Sparse Polynomial Pseudo Division Using a Heap

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 pseudo-division of sparse multivariate polynomials with integer coefficients. It uses a heap of pointers to simultaneously merge the dividend and partial products, sorting the terms efficiently and delaying all coefficient arithmetic to produce good complexity. The algorithm uses very little memory and we expect it to run in the processor cache. We give benchmarks comparing our implementation to existing computer algebra systems.

1. INTRODUCTION

Polynomial arithmetic is an essential feature of computer algebra systems. This paper examines the division of multivariate polynomials with rational coefficients. First observe that there are $\binom{n+d}{d}$ monomials in n variables of degree $\leq d$. Thus as n and d increase, polynomials must be sparse if they are to fit in computer memory.

The best representation for sparse polynomials is likely to depend on the application. For example, a recursive dense format permits the use of asyptotically fast algorithms [12], and is well suited to GCD and factorization algorithms which construct the result one variable at a time [6].

This paper is about the sparse distributed representation, which is commonly used at the top level of computer algebra systems. Users enter polynomials in this format and they write programs using it, expecting the cost of arithmetic to reflect the work they would do by hand.

Algorithms for the sparse distributed representation have a distinctly classical feel. Their performance depends on how you sort terms and do coefficient arithmetic. On a modern computer, cache memory also critically affects performance. Division in particular has been a frequent bottleneck [6, 8].

Our paper is organized as follows. In §2 we analyze the problem and present our strategy for performing arithmetic. In §3 we present the heap algorithms for division and our new algorithm for pseudo division. We also discuss optimizations and issues of performance. In §4 we present a series of benchmark problems.

2. SPARSE POLYNOMIAL DIVISION

Consider the problem of dividing $f \in \mathbb{Z}[x_1, \ldots, x_n]$ by $g \in \mathbb{Z}[x_1, \ldots, x_n]$, producing a quotient q and a remainder r in $\mathbb{Q}[x_1, \ldots, x_n]$. We will assume that the polynomials are stored in a sparse distributed format that is sorted with respect to a monomial order $<$.

Starting with $q = r = 0$, each division step examines the next non-zero term of $f - qg - r$ in descending order by $<$. Call this current term t. When t is divisible by g_1, we add a new term t/g_1 to the quotient q to cancel t using $-qg$. Otherwise we move t to the remainder r.

How we compute t does not affect the correctness of the division algorithm, but it will determine the performance. There are two tasks to perform: the terms of $f - qg - r$ must be sorted, and the coefficients of any equal monomials must be added up.

2.1 Monomial Comparisons

First consider how to sort the terms of $f - qg - r$ that are generated in a division. There are $\#f + \#q(\#g - 1)$ terms in total, since the terms of $-qg_1$ are constructed to cancel something and the terms of r come from $f - qg$. We will use a comparison sort, however trie-based digital sorts have also been used [4].

The naive approach to sorting the terms is to merge them into an ordered linear structure, such as a linked list or a dynamic array. This is the classical algorithm for division: create an intermediate polynomial $p := f$, and when a new term of the quotient q_i is computed subtract $q_i g$ from p using a merge. Unfortunately each merge is $O(\#p + \#g)$. This means, for example, that an exact division with $\#q = \#g = 1000$ and $\#f = 10^6$ can do $O(10^9)$ comparisons.

The problem of too many comparisons was noticed by Johnson [9], whose "quotient heap" algorithm we present in §3. Another solution is the "geobucket" data structure of Yan [15]. It uses buckets with $\{4, 8, 16, \ldots, 2^i\}$ terms and merges each polynomial with the first bucket that is larger. If the result is too large to store in that bucket then it is merged with the next bucket, and so on, until the sum can be stored. The division algorithm with a geobucket does $O(N \log N)$ comparisons where $N = \#f + \#q(\#g - 1)$ [15]. The actual number of comparisons is often better than a heap, but the algorithm runs slower because it uses main memory whereas a heap can fit in the cache [13].

Next we consider the minimum number of comparisons required to divide. Horowitz studied sparse multiplication and observed that products of sorted polynomials generate a rectangular *tableau* [7, 11].

Definition 1. *Let $n_1 \geq \cdots \geq n_m$ be positive integers. A* standard Young tableau *of shape $(n_1, n_2, \ldots, n_m)$ is an arrangement of the integers 1 to $n_1 + \cdots + n_m$ into m rows, where row i has n_i elements and rows and columns strictly increase.*

To construct a tableau, order the products $f_i g_j$ using the monomial order so that the largest term is assigned 1, the second largest 2, and so on. Ties can be broken in any way. We will prefer terms closer to the bottom. Note that ties can not occur in any row or column of the matrix of products if the monomials of the multiplicands are distinct.

Example 2. *Consider $(x^3 + x + 1)(x^5 + x^3 + x + 1)$. We show a product matrix of the exponents that appear and a corresponding tableau that orders the terms.*

$$\begin{array}{c|cccc} \times & x^5 & x^3 & x^1 & x^0 \\ \hline x^3 & 8 & 6 & 4 & 3 \\ x^1 & 6 & 4 & 2 & 1 \\ x^0 & 5 & 3 & 1 & 0 \end{array} \qquad \begin{array}{|cccc|} \hline 1 & 3 & 6 & 8 \\ 2 & 5 & 9 & 11 \\ 4 & 7 & 10 & 12 \\ \hline \end{array}$$

product matrix *tableau*

Sorting an $n \times m$ tableau requires $O(nm \log(\min(n, m)))$ comparisons [5], which is achieved by a divide-and-conquer merge or a simultaneous $min(n, m)$-ary merge of the rows or columns. Surprisingly, this may not be a lower bound for sorting the product matrix. That problem is "X+Y sorting", and finding a faster algorithm for it is an open problem [2].

Due to this uncertainty, we will state the complexity of division in terms of multiplication. Let $M(n, m)$ be a lower bound on the number of comparisons needed to multiply two sparse polynomials with n and m terms. The product could have nm terms so we will assume $nm \leq M(n, m)$.

A sparse division $f \div g = (q, r)$ multiplies the quotient q by $g - LT(g)$, producing up to $\#q(\#g - 1)$ terms. These terms must be merged with the dividend f, since only one term of f must be divisible and the rest could be anything. This merge requires $O(\#f + \#q(\#g - 1))$ comparisons, so the number of comparisons required for a sparse division is $O(\#f + M(\#q, \#g - 1))$.

2.2 Coefficient Arithmetic

We now consider the arithmetic that must be performed. Multiplying sparse polynomials with n and m terms requires nm coefficient multiplications, but the number of additions depends on the number of equal monomials. This is limited by the product matrix structure which can not have ties in any row or column. The most ties occur when the monomials in each "slice" $\{a_{ij} \mid i + j = c\}$, $c \in \{2, ..., n + m\}$ are equal. This is what happens in a dense univariate multiplication, but it can happen even when the polynomials are not dense. One term from each slice is not added to anything, so at most $nm - n - m + 1$ additions are required.

Consider a sparse division $f \div g = (q, r)$ that runs over $\mathbb{Z}$. Merging $-q(g - LT(g))$ with f does at most $\#f$ additions, and a division is needed to construct each coefficient of q. A total of $\#q(\#g - 1)$ multiplications, $\#q$ divisions, and up to $\#f + (\#q - 1)(\#g - 2)$ additions are performed, which is comparable to a sparse polynomial multiplication.

Now suppose the quotient q has rational coefficients and we use fractions for the arithmetic. Each of the $\#q(\#g - 1)$ products now multiplies a fraction by an integer, which we would implement as follows:

```
# compute (a/b)*c
mulqz(numerator a, denominator b, integer c)
  g := gcd(b,c);
  if (g = 1) then
    A := a*c;
    return (A, b);
  else
    B := b/g;
    C := c/g;
    A := a*C;
    return (A, B);
end;
```

The cost of mulqz is easy to see. Note that all divisions are exact. To construct each term of the quotient we divide a fraction by an integer, calling mulqz with the numerator and denominator swapped. To add fractions we will use the grade-school method: $a/b + c/d = (ad + bc)/(bd)$ followed by a gcd and two exact divisions. The alternatives in [10] can be faster, but they require more operations.

Using our formulas for sparse division in the integer case, we computed the number of integer operations performed in a division over $\mathbb{Q}$ when fractions are used. The results are summarized in Table 1.

Table 1: Operations in $\mathbb{Z}$ when fractions are used

multiplications	$3f + 3(q-1)(g-1) + qg$
exact divisions	$2f + 2(q-1)(g-1) + 2qg$
gcds	$f + (q-1)(g-1) + qg$
additions	$f + (q-1)(g-1)$

A quick estimate says that if multiplications, divisions, and gcds are about the same cost then we will do ten times more arithmetic using fractions than we did over $\mathbb{Z}$. When $\#f = \#q\#g$ it will be sixteen times more. Note that most of this cost is from adding fractions. The costs from mulqz are isolated in the table. They are qg, $2qg$, qg, and 0.

The process of *pseudo division* was developed in response to the high cost of fraction arithmetic. Fractions are scaled to a common denominator that is updated with each new term of the quotient, so that instead of adding fractions we add integers. Let us examine this algorithm as it is typically implemented. LM(p), LC(p), and LT(p) denote the leading monomial, coefficient, and term of a polynomial p.

```
pseudo_remainder(polynomial f, polynomial g)
  p := f;
  r := 0;
  S := 1;  # common denominator
  while (p != 0) do
    if (LM(g) divides LM(p)) then
      G := gcd(LC(p), LC(g));
      A := LC(p)/G;
      B := LC(g)/G;
      S := S*B;            # update denominator
      q := LM(p)/LM(g);  # monomial quotient
      p := B*p - (A*q)*g;
    else
      r := r + LT(p);
      p := p - LT(p);
  end loop;
  return (r, S);
end;
```

The algorithm performs one gcd, two exact divisions, and one multiplication for each term of the quotient. Those are necessary to update the common denominator. The problem is $p := B * p - (A * q) * g$ and the number of multiplications this could do.

Consider an exact division $f \div g = q$ where $\#f = \#q\#g$ and the denominators of q strictly increase. In step i of this division p will have $(\#q - i + 1)\#g$ terms and we will do $(\#q - i + 2)\#g$ multiplications. In total

$$\sum_{i=1}^{\#q} (\#q - i + 2)\#g = \frac{(\#q + 3)\#q\#g}{2}$$

multiplications will be done. This is an order of magnitude more work than using fractions. A division with remainder is even worse if f has many terms that are moved to the remainder at the end. The algorithm can do $O(qf + q^2 g)$ multiplications in general when the polynomials are sparse.

For dense univariate polynomials the algorithm above is faster than using fractions. In that case $\#f = \#q + \#g - 1$ and each division step removes one term from p. The total number of multiplications is

$$\sum_{i=1}^{\#q} (\#q + 2\#g - i) = \frac{\#q(\#q + 4\#g - 1)}{2}.$$

The total amount of arithmetic is given by Table 2 below. If multiplications, exact divisions, and gcds are about the same cost and $\#q \sim \#g$, the algorithm will be roughly four times faster than using fractions. The $q^2/2$ multiplications are impractical for divisions with large quotients though.

Table 2: Classical dense pseudo division in $\mathbb{Q}[x]$

multiplications	$q(q + 4g - 1)/2 + q$
exact divisions	$2q$
gcds	q
additions	qg

The reason for the poor complexity of pseudo division is that terms are scaled multiple times before they are used. This is obvious for sparse polynomials but it appears as a $q^2/2$ term in the dense univariate case as well. This extra arithmetic is an artifact of how we sort the terms.

The correct number of integer operations is the following. Imagine we have a common denominator S and and we are adding up the coefficient of the next term into a sum A/S. To add a term of f we would compute `A += S*f_i` and to add a product $q_i g_j$ we could compute

```
A -= (S / denominator(q_i)) * numerator(q_i) * g_j
```

If the current monomial is divisible by $\mathrm{LM}(g)$ we will divide A/S by $\mathrm{LC}(g)$ with `(S, q) := mulqz(S, A, LC(g))`. This updates S and computes the numerator q of the next term of the quotient. We computed the cost of this scheme from the operation counts of the integer case. The result is Table 3. Pseudo division is about three times more work than a division over $\mathbb{Z}$ or four times more when $\#f \sim \#q\#g$.

Table 3: Operations in $\mathbb{Z}$ for pseudo division

multiplications	$f + 2q(g - 1) + q$
exact divisions	$q(g - 1) + 2q$
gcds	q
additions	$f + (q - 1)(g - 2)$

3. DIVISION USING A HEAP

Let us put it all together and develop three algorithms. We have seen that to implement pseudo division efficiently we must delay all coefficient arithmetic as long as possible. We have also seen that multiplication generates a tableau that can be sorted in $nm \log(\min(n, m))$ comparisons.

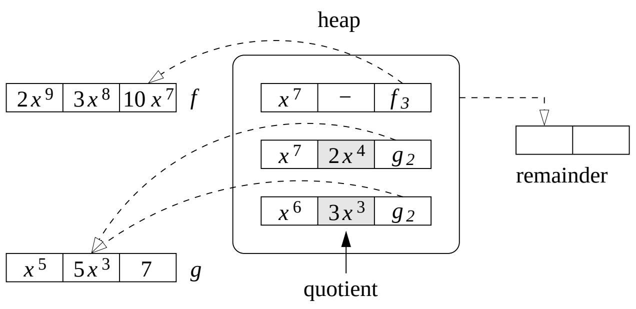
The idea of these algorithms is to simultaneously merge all of the polynomials that appear in the division. We will use a pointer into each polynomial and a binary heap. Each iteration extracts the terms with the largest monomial and adds them up. If the sum is not zero we will compute a new term of the quotient or move the term to the remainder. Finally the next term of each polynomial that was used is inserted into the heap for the next iteration.

Example 3. *Consider $f \div g$ where $f = 2x^9 + 3x^8 + 10x^7$ and $g = x^5 + 5x^3 + 7$. The quotient is $q = 2x^4 + 3x^3 - 15x$ and the remainder is $r = 61x^4 - 21x^3 + 105x$. We will write out the division and step through the algorithm.*

f	$2x^9$	$3x^8$	$10x^7$				
$-q_1 g$	$-2x^9$		$-10x^7$		$-14x^4$		
$-q_2 g$		$-3x^8$		$-15x^6$		$-21x^3$	
$-q_3 g$				$15x^6$	$75x^4$		$105x$
r					$61x^4$	$-21x^3$	$105x$

1: *We begin with $f_1 = 2x^9$ in the heap. We extract it and compute the first term of the quotient $q_1 = 2x^4$. We now merge the rest of f with $-q_1 g$. We insert $f_2 = 3x^8$ and $-q_1 g_2 = -10x^7$ into the heap for the next iteration.*
2: *The heap now contains $f_1 = 3x^8$ and $-q_1 g_2 = -10x^7$. We extract the largest term, $3x^8$, and compute $q_2 = 3x^3$. Now we are merging $\{f, -q_1 g, -q_2 g\}$. We insert $f_3 = 10x^7$ and $-q_2 g_2 = -15x^6$ into the heap.*
3: *The heap contains $f_3 = 10x^7$, $-q_1 g_2 = -10x^7$, and $-q_2 g_2 = -15x^6$. We extract the first two, sum them to zero, and insert $-q_1 g_3 = -14x^4$. We are done merging f.*
4: *The heap contains $-q_2 g_2 = -15x^6$ and $-q_1 g_3 = -14x^4$. We extract $-15x^6$ and compute $q_3 = -15x$. We insert $-q_2 g_3 = -21x^3$ and $-q_3 g_2 = 75x^4$ into the heap.*
5: *The heap contains $-q_1 g_3 = -14x^4$, $-q_3 g_2 = 75x^4$, and $-q_2 g_3 = -21x^3$. We extract the first two and compute $-14x^4 + 75x^4 = 61x^4$. This is not divisible by $\mathrm{LT}(g)$ so we move it to the remainder. We insert $-q_3 g_3 = 105x$.*
6: *The heap contains $-q_2 g_3 = -21x^3$ and $-q_3 g_3 = 105x$. We extract $-21x^3$ and move it to the remainder.*
7: *The heap contains $-q_3 g_3 = 105x$. We extract this term and move it to the remainder. Division is complete.* □

Figure 1: Example 3 step 3 in computer memory



We have described the "quotient heap" division algorithm of Johnson [9]. It divides $f \div g = (q, r)$ by using a heap with $\#q + 1$ elements to merge f with each $-q_i g$. The cost to insert and extract terms is $O(\log(\#q + 1))$ comparisons, so the total number of comparisons is

$$O((\#f + \#q(\#g - 1)) \log(\#q + 1)).$$

What makes this algorithm especially competitive is its memory requirement. It is not necessary to store each $-q_i g$, their terms may be generated as the algorithm runs if we store the right information. Each element of the heap needs a pointer to q_i and an index j into g so that we can compute $-q_i g_{j+1}$ from $-q_i g_j$ and insert the next term. The terms of f may be distinguished by a null pointer in place of q_i. This design is illustrated in Figure 1.

The total amount of memory required is $O(\#q + \#r)$, which is highly desirable since it is the size of the result. By comparison, geobuckets use $O(N \log N)$ storage where $N = \#f + \#q(\#g - 1)$. This order of magnitude savings allows the algorithm to run entirely in the processor cache provided the quotient is not too large.

3.1 The Divisor Heap Algorithm

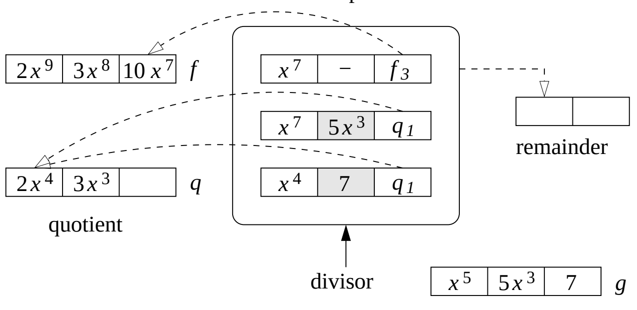
When the quotient is large and the divisor is small the algorithm above is not very efficient. There are important cases where this occurs. For example, if we compute $g = \gcd(a, b)$ and divide a/g and b/g to recover the cofactors, the divisor g will often be much smaller than the quotients. This is actually a bottleneck in gcd algorithms [6, 8].

The divisor heap algorithm of Monagan and Pearce [13] lowers the complexity of these divisions by using a heap to merge f with each $-g_i q$. That is, heap elements increment along the quotient and multiply by a term of the divisor instead of the reverse. There are $\#g$ elements in the heap so the total number of comparisons is

$$O((\#f + \#q(\#g - 1)) \log(\#g)).$$

A subtlety in the algorithm is that we may merge $-g_i q_j$ before q_{j+1} has been computed. In that case we can not compute $-g_i q_{j+1}$ and insert it into the heap immediately, we must wait until q_{j+1} is known. Our solution is to keep track of which g_i have a product in the heap. After $-g_i q_j$ is extracted from the heap and we go to insert $-g_i q_{j+1}$, we check whether g_{i+1} has a term in the heap. If not, and if $-g_{i+1} q_k$ can be computed (for some q_k), we compute this term and insert it into the heap.

Figure 2: Example 4 Step 3 in computer memory



Example 4. *We revisit Example 3 with the divisor heap algorithm. Let $f = 2x^9 + 3x^8 + 10x^7$ and $g = x^5 + 5x^3 + 7$. We compute $f \div g = (q, r)$ where $q = 2x^4 + 3x^3 - 15x$ and $r = 61x^4 - 21x^3 + 105x$. The shaded row $-g_1 q$ shows the cancellations that occur, its terms are not in the heap.*

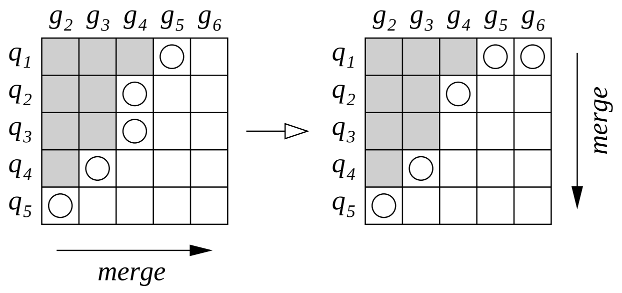
f	$2x^9$	$3x^8$	$10x^7$				
$-g_1 q$	$-2x^9$	$-3x^8$		$15x^6$			
$-g_2 q$			$-10x^7$	$-15x^6$	$75x^4$		
$-g_3 q$					$-14x^4$	$-21x^3$	$105x$
r					$61x^4$	$-21x^3$	$105x$

1: *Initially $k = 3$ and $f_1 = 2x^9$ is in the heap. We extract $f_1 = 2x^9$ and compute $q_1 = 2x^4$. We insert $-g_2 q_1 = -10x^7$ and f_2 into the heap.*
2: *We extract $f_2 = -3x^8$, compute $q_2 = 3x^3$, and insert f_3.*
3: *We extract $f_3 = 10x^7$ and $-g_2 q_1 = -10x^7$ and get zero. We insert $-g_2 q_2 = -15x^6$ and $-g_3 q_1 = -14x^4$.*
4: *We extract $-g_2 q_2 = -15x^6$ and compute $q_3 = -15x$. We insert $-g_2 q_3 = 75x^4$ and check for a product involving g_3.*
5: *We extract $-g_2 q_3 = 75x^4$ and $-g_3 q_1 = -14x^4$ and move $61x^4$ to the remainder. We insert $-g_3 q_2 = -21x^3$.*
6: *We extract $-g_3 q_2 = -21x^3$ and move it to the remainder. We insert $-g_3 q_3 = 105x$.*
7: *We extract $-g_3 q_3 = 105x$ and move it to the remainder. Now the heap is empty so division is complete.* □

3.2 Minimal Heap Division

We now have two heap algorithms to divide $f \div g = (q, r)$ that add a factor of $\log(\#q + 1)$ or $\log(\#g)$ to the number of comparisons. However we may not know what algorithm to run because the size of the quotient is a priori unknown. Instead we will dynamically switch from a quotient heap to a divisor heap in the middle of a computation. This will ensure that we always have the best bound on the number of monomial comparisons.

Figure 3: Switching from quotient to divisor heap


g_2 g_3 g_4 g_5 g_6
q_1 q_2 q_3 q_4 q_5
merge
g_2 g_3 g_4 g_5 g_6
q_1 q_2 q_3 q_4 q_5
merge

A general instance of this problem is shown in Figure 3. The shaded regions show what products have been merged. On the left is a quotient heap with $-q_i g_j$ for all $1 \le i \le n$. This is a covering set for the tableau with one element in each row [7]. Our task is to compute the covering set on the right that has one element in each column. We store the row set as a list of indices into g, that is $R = [5, 4, 4, 3, 2]$. The column set $C = [5, 4, 2, 1, 1]$ is the list of indices into q that we must compute.

We give a simple algorithm that is linear time. Starting from the bottom left corner of the row set, we search up through each column to find the smallest row index and assign this index to the column set. The tableau structure implies that the algorithm only needs to move up and right.

```
# compute a column set from a row set
# R has elements 1..n
column_set(array R, integer n)
  C := array(1..n);
  i := n;
  j := 1;
  while (j <= n) do
    while (i > 1 and R[i-1] = j+1) do
      i := i-1;
    end loop;
    C[j] := i;
    j := j+1;
  end loop;
  return C;
end;
```

With the algorithm above, we can switch from a quotient heap to a divisor heap when $\#q + 1 = \#g$ to guarantee minimality. Although the heap is destroyed by the switch, it can be rebuilt in linear time [11]. This approach does not adapt very well to divisions with multiple divisors though. The heap may be destroyed and rebuilt multiple times, and the resulting algorithm is complicated.

We give a simpler algorithm that does not destroy the heap and does $O(\#f + \#q(\#g-1)\log(\min(\#q, \#g-1)))$ comparisons. Our benchmarks show that it is as fast as a multiplication.

The algorithm uses a heap with two types of products in it. We say that a product $q_i g_j$ "moves along g" if the next term we insert into the heap is $q_i g_{j+1}$. This corresponds to products in the quotient heap algorithm. Likewise, $q_i g_j$ "moves along q" if the next term is $q_{i+1} g_j$, like in the divisor heap algorithm.

The new algorithm begins by adding products $q_i g_2$ that move along g each time a term of the quotient is computed. When $\#q = \#g - 1$, it initializes indices ($= \#q$) for each g_j into q and adds $q_{\#q} g_2$ to the heap. This product and subsequent products move along q starting from that term.

We now present the algorithm. Its cost in comparisons and arithmetic operations is given by Theorem 5.

Theorem 5. *Algorithm 1 divides f by g using at most $O(\#f + \#q(\#g-1)\log(\min(\#q, \#g-1)))$ comparisons. If the denominator $s = 1$ at most $\#q(\#g-1)$ multiplications and $\#q$ divisions in $\mathbb{Z}$ are performed. Otherwise the maximum number of operations in $\mathbb{Z}$ is given by Table 3.*

Proof. If $\#q < \#g - 1$ the maximum size of the heap is $\#q$, otherwise it is $2(\#g-1)$. $O(\log(\min(\#q, \#g-1)))$ comparisons are needed to insert and extract the terms of $q(g - LT(g))$. An additional $\#f + \#q(\#g-1)$ comparisons are required to merge f with these terms. In total we do $O(\#f + \#q(\#g-1)\log(\min(\#q, \#g-1)))$ comparisons.

If $s = 1$ all $denom(q_i) = s$ and we don't scale terms of f. $LC(g) \mid c$ always succeeds, and $\#q(\#g-1)$ multiplications and $\#q$ divisions are performed in total.

When $s > 1$ the strategy is described at the end of §2.2. Each of the $\#q(\#g-1)$ terms in the heap is scaled using an exact division and two multiplications. One multiplication is used to scale terms of f. There are $\#q$ extra divisions (not exact) for testing $LC(g) \mid c$. If these return a remainder it should be used to speed up the mulqz gcd. Finally mulqz performs a gcd, two exact divisions, and one multiplication for each term of the quotient q. □

Algorithm 1: Minimal Heap Pseudo Division.
Input: $f, g \in \mathbb{Z}[x_1, ..., x_n]$, $g \neq 0$, a monomial order $<$.
Output: $q, r \in \mathbb{Q}[x_1, ...x_n]$ with $f = qg + r$ and no term of r divisible by $LT(g)$.

```
(q, r, s) := (0, 0, 1).
H := empty heap ordered by <.
k := 1.
while (|H| > 0 or k ≤ #f) do
  if (k ≤ #f and (|H| = 0 or LM(f_k) ≥ LM(H_1)))
    m := LM(f_k).
    if (s = 1)
      c := LC(f_k).
    else
      c := s * LC(f_k).
    k := k + 1.
  else
    m := LM(H_1).
    c := 0.
  while (|H| > 0 and LM(H_1) = m) do
    extract H_1 = q_i g_j from the heap.
    if (denom(q_i) = s)
      c := c − LC(numer(q_i)) * LC(g_j).
    else
      c := c − (s/denom(q_i)) * LC(numer(q_i)) * LC(g_j).
    if (q_i g_j moves along g and j ≤ #g)
      insert q_i g_{j+1} into H moving along g.
    else if (i < #q)
      insert q_{i+1} g_j into H moving along q.
      increment the index of g_j into q.
    if (q_i g_j moves along q and j < #g)
      t := the index of g_{j+1} into q.
      if (t ≤ #q and q_t g_{j+1} ∉ H)
        insert q_t g_{j+1} into H moving along q.
  end loop.
  if (c ≠ 0 and LM(g) | m)
    if (LC(g) | c)
      c := c/LC(g).
    else
      (s, c) := mulqz(s, c, LC(g)).
    q := q + (c/s)(m/LM(g)).
    if (#q < #g − 1)
      insert q_(#q) g_2 into H moving along g.
    else if (#q > #g − 1)
      t := the index of g_2 into q.
      if (q_t g_2 ∉ H)
        insert q_t g_2 into H moving along q.
    else
      set the index of each g_j into q to be #q.
      insert q_(#q) g_2 into H moving along q.
  else if (c ≠ 0)
    r := r + (c/s) * m.
end loop.
return (q, r)
```

3.3 Optimizations

Although Theorem 5 provides some basic performance guarantees, we have found many optimizations that greatly improve the real world performance of Algorithm 1.

The most important optimization is to implement the heap efficiently. The best known algorithm for shrinking a heap uses twice as many comparisons as another classical algorithm described in [13, 14].

Another optimization described in [13] is chaining equal elements in the heap. This is needed to reduce the number of comparisons in the dense case to linear. That is, a dense multiplication of $n \times m$ terms does $O(nm)$ comparisons.

For this paper we made a third optimization to the heap. Monomials are stored directly in the heap if they are one word long. This greatly improves memory locality and helps the compiler to do more optimizations. Disabling this slows down our program by 30-50%.

Another important optimization is to add up products of word-sized integers separately. We wrote some assembly code to avoid calling multiprecision routines. Disabling this code slows down our program by 30-50%. It adds precisely 50 seconds to Fateman's benchmark.

A significant feature of our software is the ability to pack multiple exponents into each machine word. The details are presented in [13]. Decreasing the size of monomials speeds everything up and lets us do larger problems. For example, the second benchmark has a polynomial with 20M terms. With no packing it takes 1.6GB. With packing it is 300MB. Packing exponents allows us to handle polynomials with up to 500M terms in 16 GB of RAM.

We end with two remarks about why the heap algorithms are fast. Fateman studied sparse polynomial multiplication in [3] and found that performance was critically affected by locality and cache. In particular: traversing large structures was slow, randomly accessing them was very slow, creating many intermediate multiprecision integers was slow, and it had poor locality which made integer operations very slow. It also necessitated garbage collection, which was slow.

The heap algorithms do none of those things. An $n \times m$ multiplication randomly accesses $O(min(n, m))$ memory. If one polynomial is large, its terms are accessed in $min(n, m)$ simultaneous passes. No structures are used other than the heap, and for all feasible computations it fits in the cache. Terms are added up and written out in order, so only one multiprecision integer is needed to run the whole algorithm. No "garbage" is created at all. Finally, the multiprecision coefficients of a polynomial can be stored in a second array in order, ensuring good locality for them as well.

4. BENCHMARKS

We ran benchmarks using one core of an Intel Xeon 5160 (Core 2 Duo) 3.0 GHz with 4 MB of L2 and 16 GB of RAM, running in 64 bit mode with GMP 4.2.1. Our software is a C library that uses heaps of pointers to compute with sparse polynomials [13]. We give two times for our library. In the slow time we store each exponent in a 64 bit integer. For the fast time we pack all of the exponents into one 64 bit integer and use word operations to compare, multiply, and divide monomials.

4.1 Fateman's Benchmark

Our first benchmark is due to Fateman [3], who compared the routines for sparse polynomial multiplication in different computer algebra systems and offered insights into how cache and memory access can affect performance.

Let $f = (1 + x + y + z + t)^{30}$ and $g = f + 1$. We multiply $p = f * g$ and divide $q = p/f$. The polynomials f and g have 46376 terms and 61 bit coefficients. The product p has 635376 terms and 128 bit coefficients. We use lexicographic order with $x > y > z > t$.

Table 4: Dense multiplication and division over $\mathbb{Z}$

$46376 \times 46376 = 635376$	$p = f * g$	$q = p/f$
heap (1 word monomial)	80.480	101.490
heap (4 word monomial)	242.870	236.600
Trip v0.98.84 (floating point)	76.161	-
Trip v0.98.84 (rationals)	352.045	-
Magma V2.14-7	679.070	610.620
Singular 3-0-4	1482.360	364.490

Fateman's benchmark is a dense computation. The $\binom{n+d}{d}$ monomials of degree $\leq d$ are multiplied to produce a result with $\binom{n+2d}{2d}$ terms. We could expect up to $\binom{n+d}{d}^2$ terms if the problem were sparse.

As a measure of sparsity, we introduce the *dispersion* D of a multiplication as the size of the result divided by the size of each multiplicand. Then $1/D$ is the average number of terms added to produce each term of the result. For the problem above $D = 635376/46376^2 = .0295\%$.

4.2 Sparse Problems in Many Variables

Our next benchmark is a sparse computation in many variables. For (n, d) let $f = (x_1x_2 + x_2x_3 + \cdots + x_nx_1 + \sum_{i=1}^{n} x_i + 1)^d$ and $g = (\sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n} x_i + 1)^d$. We multiply $p = f * g$ and divide $q = p/f$.

We will use $n = 10$ and $d = 5$. Then f has 26599 terms, g has 36365 terms, and p has 19631157 terms. Although this is a sparse problem, the dispersion is only 2%. We will use lexicographic order with $x_1 > x_2 > \cdots > x_{10}$.

Table 5: Sparse multiplication and division over $\mathbb{Z}$

$26599 \times 36365 = 19631157$	$p = f * g$	$q = p/f$
heap (1 word monomial)	49.090	52.580
heap (10 word monomial)	221.910	208.280
Trip v0.98.84 (floating point)	91.573	-
Trip v0.98.84 (rationals)	259.408	-
Magma V2.14-7	313.020	5744.600
Singular 3-0-4	655.250	206.600

We managed to beat Trip on this benchmark despite the fact that it uses a burst-trie to sort the terms in linear time. The difference in speed is due to caching effects [3]. The heap algorithms write out the result one term at a time while accessing $O(\#f + \#g)$ memory. Trip (and other systems) add terms to an intermediate structure that is $O(\#p)$. If the result is large this structure will not fit in the cache, and memory access penalties (200 cycles) will be incurred. The heap algorithms in general do not incur these penalties.

4.3 Unbalanced Divisions (Very Sparse)

Our next benchmark is extremely sparse, and tests the efficiency of sorting as the quotient and divisor vary in size.

Let $f = (1 + x + y + 2z^2 + 3t^3 + 5u^5)^{12}$ and $g = (1 + u + t + 2z^2 + 3y^3 + 5x^5)^{12}$. We multiply $p = f * g$ and divide $q = p/f$. f and g have 6188 terms and 37 bit coefficients. p has 5821335 terms and its coefficients are 75 bits. The dispersion $D = 15.2\%$.

Next we use different powers $\{4, 8, 12, 18, 30\}$ in f and g to vary the size of the quotient and the divisor. We report the number of divisor terms $\#f$ and quotient terms $\#g$, and the time for multiplication $p = f * g$ and division $q = p/f$. We do not count the time to export the result of either computation. We conclude that our minimal heap division sorts the terms as efficiently as a multiplication.

Table 6: Sparse multiplication and division over $\mathbb{Z}$

$6188 \times 6188 = 5821335$	$p = f * g$	$q = p/f$
heap (1 word monomial)	2.910	2.240
heap (5 word monomial)	7.540	5.460
Trip v0.98.84 (floating point)	2.854	-
Trip v0.98.84 (rationals)	8.369	-
Magma V2.14-7	23.770	151.990
Singular 3-0-4	58.910	39.250

Table 7: Varying the quotient and the divisor

powers		$\#f$	$\#g$, $\#q$	$p = f * g$	$q = p/f$
30	4	324632	126	2.990	2.770
18	8	33649	1287	2.270	2.220
12	12	6188	6188	2.440	2.240
8	18	1287	33649	2.380	2.460
4	30	126	324632	2.840	2.530

4.4 Pseudo Remainder

Our final problem is a pseudo-remainder computation. Let $f = ((x-2)(2y-3)(3z-5)(5t-7)(7u-11)(11v-13))^8$ and $g = ((2x-1)(3y-2)(5z-3)(7t-5)(11u-7)(13v-11))^4$. We compute the remainder of f divided by g using grevlex order with $x > y > z > t > u > v$. This order uses one extra word per monomial in our software.

f has 531441 terms and 143 bit coefficients. The divisor has 15625 terms and 69 bit coefficients. The quotient has 15625 terms and 132 bit numerators, and the remainder has 515816 terms and 206 bit numerators. The denominator for the quotient and remainder is 9480443485314601800, which is 64 bits.

Table 8: Pseudo-remainder in $\mathbb{Q}[x, y, z, t, u, v]$

$531441 \div 15625 = 15625 + 515816$	$f/g = (q, r)$
heap (2 word monomial)	31.170
heap (7 word monomial)	44.780
Magma V2.14-7	225.260
Singular 3-0-4	3686.870

5. CONCLUSIONS

We analyzed sparse polynomial division and presented a new algorithm for pseudo division. It does less arithmetic than previous algorithms and it sorts terms as efficiently as a divide-and-conquer multiplication. Its space requirements are linear in the size of the input and the result, allowing it to run inside the cache of modern processors. We presented benchmarks showing that the algorithm achieved very good performance on a variety of problems.

6. REFERENCES

[1] O. Bachmann, H. Schönemann. Monomial representations for Gröbner bases computations. *Proceedings of ISSAC 1998*, ACM Press (1998) 309–316.

[2] E. Demaine, J. Mitchell, J. O'Rourke. Problem 41: Sorting X+Y (Pairwise Sums). *The Open Problems Project*, `http://maven.smith.edu/~orourke/TOPP/P41.html`

[3] R. Fateman. Comparing the speed of programs for sparse polynomial multiplication. *ACM SIGSAM Bulletin*, **37** (1) (2003) 4–15.

[4] M. Gastineau, J. Laskar. Development of TRIP: Fast Sparse Multivariate Polynomial Multiplication Using Burst Tries. *Proceedings of ICCS 2006*, Springer LNCS 3992 (2006) 446–453.

[5] L.H. Harper, T.H. Payne, J.E. Savage, E. Straus. Sorting X+Y. *C. ACM*, **18** (6) (1975) 347–349.

[6] M. van Hoeij, M. Monagan. A Modular GCD Algorithm Over Number Fields Presented With Multiple Field Extensions. *Proceedings of ISSAC 2002*, ACM Press (2002) 109–116.

[7] E. Horowitz. A Sorting Algorithm for Polynomial Multiplication. *J. ACM*, **22** (4) (1975) 450–462.

[8] S. Javadi, M. Monagan. A Sparse Modular GCD Algorithm for Polynomials Over Algebraic Function Fields. *Proceedings of ISSAC 2007*, ACM Press (2007) 187–194.

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

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

[11] D. Knuth. The Art of Computer Programming, Volume 3: *Sorting and Searching.* Additson-Wesley (1998).

[12] X. Li, M. Maza, E. Schost. Fast Arithmetic For Triangular Sets: From Theory To Practice. *Proceedings of ISSAC 2007*, ACM Press (2007) 269–276.

[13] M. Monagan, R. Pearce. Polynomial Division Using Dynamic Arrays, Heaps, and Packed Exponent Vectors. *Proceedings of CASC 2007*, Springer (2007) 295–315.

[14] R. Pearce. How to Implement Binary Heaps. `http://www.cecm.sfu.ca/~rpearcea/heaps.html`

[15] T. Yan. The geobucket data structure for polynomials. *J. Symb. Comput.* **25** (1998) 285–293.