Polynomial Division using Dynamic Arrays, Heaps, and Packed Exponent Vectors *

Michael Monagan and Roman Pearce

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

Abstract. A common way of implementing multivariate polynomial multiplication and division is to represent polynomials as a linked list of terms sorted in a term ordering and to use repeated merging. This results in poor performance on large sparse polynomials.

In this paper we use an auxiliary heap of pointers to reduce the number of monomial comparisons in the worst case while keeping the overall storage linear. We give two variations. In the first, the size of the heap is bounded by the number of terms in the quotient(s). In the second, which is new, the size is bounded by the number of terms in the divisor(s).

We use dynamic arrays of terms rather than linked lists of terms to reduce storage allocations and indirect memory references. We pack monomials in the array to reduce storage and speed up monomial comparisons. We give a new packing for the graded reverse lexicographical monomial ordering.

We have implemented the heap algorithms in C with an interface to Maple. For comparison we have also implemented Yan's "geobuckets" data structure. Our timings demonstrate that heaps of pointers are comparable in speed with geobuckets but use significantly less storage.

1 Introduction

In this paper we present and compare algorithms and data structures for polynomial division in the ring $P = F[x_1, x_2, ..., x_n]$ where F is a field. We are interested in (i) exact division of $f \in P$ by a single polynomial $g \in P$, that is testing whether $g|f$ and if so, computing the quotient $q = f/g$, (ii) exact division of $f \in P$ by a polynomial $g \in P$ modulo a triangular set of polynomials in $F[x_i, x_{i+1}, ..., x_n]$, and (iii) computing the remainder of $f \in P$ divided by a set of polynomials $\{g_1, g_2, ..., g_s\} \in P$. Since many algorithms in computer algebra use modular methods for efficiency, that is, they compute modulo primes, we will want to divide over characteristic p as well as characteristic 0.

We consider distributed polynomial representations that sort the terms of the polynomial with respect to a monomial ordering. See (3) or (4) for background

* This work was supported by NSERC of Canada and the MITACS NCE of Canada

material on monomial orderings. The orderings that we are most interested in are the pure lexicographical ordering (lex), the graded lexicographical ordering (grlex), and the graded reverse lexicographical ordering (grevlex). In the grlex ordering one first sorts terms by total degree and then by lexicographical order. For example, the polynomial

$$-9x^4 - 7x^3yz + 6x^2y^3z + 8y^2z^3$$

when written with terms in descending grlex order with $x > y > z$ is

$$6x^2y^3z - 7x^3yz + 8y^2z^3 - 9x^4.$$

The data structure used to represent polynomials will have a direct impact on the efficiency of the division algorithm. The data structure used by the Axiom computer algebra system (7) for Gröbner basis computations is the SDMP (Sparse Distributed Multivariate Polynomial) data structure. This is a linked list of terms where each term is a pair (c, e), where c is a (pointer to) a coefficient and e is a pointer to the exponent vector, which is an array of machine integers. Using $\langle a, b, c, ...\rangle$ to denote an array, $[a, b, c, ...]$ to denote a linked list, and (c, e) to denote a pair of pointers, the polynomial above would be represented as

$$[\,(6, \langle 2, 3, 1\rangle),\ (-7, \langle 3, 1, 1\rangle),\ (8, \langle 0, 2, 3\rangle),\ (-9, \langle 4, 0, 0\rangle)\,].$$

We recall the division algorithm. Following the notation of Cox, Little, and O'Shea (3), we let $LT(f)$, $LM(f)$, and $LC(f)$ denote the leading term, the leading monomial, and the leading coefficient of a polynomial f, respectively. These depend on the term ordering and satisfy $LT(f) = LC(f)LM(f)$.

The Division Algorithm.
Input: $f, g_1, g_2, ..., g_s \in F[x_1, ..., x_n], F$ a field.
Output: $q_1, q_2, ..., q_s, r \in F[x_1, ...x_n]$ satisfying $f = q_1g_1 + q_2g_2 + ... + q_sg_s + r$.
1: Set $(q_1, q_2, ..., q_s) := (0, 0, ..., 0)$.
2: Set $p := f$.
3: While $p \neq 0$ do
  4: Find the first g_i s.t. $LM(g_i)|LM(p)$.
  5: If no such g_i exists then set $r := r + LT(p)$ and $p := p - LT(p)$
  6: else set $(q_i, p) := (q_i + t, p - t \times g_i)$ where $t = LT(p)/LT(g_i)$.
7: Output $(q_1, q_2, ..., q_s, r)$.

Remark: If one wishes to test if $(g_1, ..., g_s)|f$ with 0 remainder then Step 5 should be modified to stop execution and output false.

If polynomials are represented as linked lists of terms, sorted in descending order in the term ordering, then accessing the leading term $LT(f)$ takes constant time, the operation $p - LT(p)$ (link to the remaining terms of p) is constant time and $r + LT(p)$ can be done in constant time by maintaining a pointer to the last term of r. The most expensive step is the subtraction $p - t \times g_i$. This requires a "merge" – one simultaneously walks down the linked list of terms in p and the linked list of terms in g_i comparing monomials. In the worst case the merge must walk to the end of both p and g_i.

1.1 Storage management and non-local memory references.

There are two sources of inefficiency in the division algorithm when the SDMP data structure is used. The first is the many intermediate pieces of storage that need to be allocated when we multiply $t\,g_i$, for example, storage for new exponent vectors in $t\,g_i$. The second is the memory references that occur during the merge when we walk down the linked lists and, for each term, link to the exponent vectors to compare monomials. These memory references cause a loss in efficiency when the polynomials are too large to fit inside the computer's cache. On a 2.4 GHz AMD Opteron 150 with 400 MHz RAM we measured the loss of speed at a factor of 6.

These two problems can be eliminated by representing polynomials as arrays with the coefficients and exponents stored in place. For example, $6x^2y^3z-7x^3yz+8y^2z^3-9x^4$ could be stored as

$$\langle\ 6,2,3,1,-7,3,1,1,8,0,2,3,-9,4,0,0\ \rangle.$$

The difference $p - tg_i$ can be computed efficiently using merging by using two arrays: one, p, that we are copying terms out of, and another, p', that we are forming the difference $p-t\times g_i$ inside. When the merge is complete we interchange the roles of p and p' for the next iteration of the division algorithm. If p' is too small to store all of the terms of p and $-t \times g_i$ we allocate a new p' with 50% more terms than are needed to reduce the chance of another allocation in the future.

These gains are partly offset by a loss of efficiency; instead of copying pointers (one word) we must now copy exponent vectors (n words). This loss can be reduced by packing multiple exponents into each word. For example, Macaulay (5) uses dynamic arrays and packed exponent vectors. Macaulay identifies the monomials $1, z, y, x, z^2, zy, y^2, zx, yx, x^2, ...$ with non-negative integers $0, 1, 2, 3, 4, 5, ...$ to encode each monomial as a machine integer. The polynomial $6x^2y^3z-7x^3yz+8y^2z^3-9x^4$ would be represented as an array of 8 words

$$\langle\ +6,63,-7,49,+8,36,-9,33\ \rangle$$

This encoding gives a very compact representation with fast monomial comparisons, but monomial multiplication and division are slow. In (1), Bachmann and Schönemann compare different monomial packings including the Macaulay encoding. They show that packing exponent vectors produces a modest speedup (a factor of 1.5 to 2) for Gröbner basis computations modulo a machine prime with the SDMP data structure. They also show that simpler packing schemes are more efficient overall than the Macaulay encoding.

1.2 The problem of too many monomial comparisons.

When using merging to subtract $p - tg_i$, a serious inefficiency may occur when $\#p$, the number of terms in p, is much larger than $\#g_i$, the number of terms in a divisor g_i. Consider $g = (x + 1)$, $q = y^n + ... + y^2 + y$ and let $p = gq =$

$xy^n + ... + x + y^n + ... + y$. If we compute p by adding $x\,q$ to q using merging, the merge does n comparisons which is efficient. In dividing f by g the first quotient is y^n and we subtract $y^n\,g = xy^n + y^n$ from $p = xy^n + ... + xy + y^n + ... + y$. The merge does n comparisons to find y^n in p. The full division does n such merges so the total number of comparisons is $O(n^2)$, much worse than multiplication.

One solution is to represent the polynomial p as a binary search tree. Then $\mathrm{LT}(p)$ can be computed with $O(\log \#p)$ monomial comparisons and the difference $p - tg_i$ can be computed with roughly $O(\#g_i \log \#p)$ comparisons. However binary trees suffer from the same cache performance problems as linked lists.

A very nice solution is the "geobucket" data structure of Yan (12), which is used by the Singular (6) computer algebra system and others. Geobuckets are described in detail in Section 2. In the geobucket data structure a polynomial p with $\#p$ terms is represented by an array of $O(\log \#p)$ "buckets" where the i'th bucket p_i is a linked list of at most 2^i terms. To subtract $t \times g_i$ from p one subtracts $t \times g_i$ from the i'th bucket of p where $2^{i-1} < \#g_i \leq 2^i$. Subtraction is done by merging two linked lists. The idea is that asymptotic efficiency is not lost when we merge two linked lists with a similar number of terms, e.g., their length differs by at most a factor of two.

In this paper we use auxiliary an "heap of pointers" instead. When dividing p by $\{g_1, g_2, ..., g_s\}$ we maintain a heap of pairs with quotient terms and pointers back into the divisors $\{g_1, g_2, ..., g_s\}$. The pointers indicate which terms have yet to be multiplied and subtracted from p.

Suppose we are dividing f by g. Let $f = gq + r$ where q is the quotient and r the remainder. With geobuckets, division does $O(\#g\,\#q(\log \#g + \log \#q))$ comparisons (12). If we use a heap, division does $O(\#g\,\#q \log \#q)$ comparisons. A second key advantage of using a heap is that it requires only $O(\#q)$ space, and, if we need to compute the remainder, $O(\#r)$ space to write down the remainder. By comparison, the simple merge and geobucket algorithms may require $O(\#g\,\#q + \#r)$ space. The main disadvantage of using a heap is that for dense polynomials the merge and geobucket algorithms are better; they do only $O(\#g\,\#q)$ comparisons. A third advantage of using a heap is that we delay all coefficient arithmetic until we need to do it. This can result in significant speedups when we want to test if g divides f but g does not divide f.

The idea of using a heap for sparse polynomial arithmetic was first investigated by Johnson in 1974 (8). Heaps were used in Altran (2), one of the earliest computer algebra systems. We are not aware of any other computer algebra system that has used heaps for polynomial arithmetic despite their good asymptotic performance. Heaps were not considered by Stoutemyer in (11) which, as far as we are aware, is the only systematic experiment ever done comparing different polynomial data structures on a computer algebra system's test suite.

1.3 Organization of the Paper

In Section 2 we describe how we encode and pack monomials for different term orderings. Our packing for graded reverse lexicographical order is new. In Section 3 we give the main algorithms that use heaps of pointers. Two algorithms are

presented. The first algorithm bounds the size of the heap by the number of terms in the quotients $\{q_1, q_2, ..., q_s\}$. In the second algorithm, the size of the heap is bounded by the number of terms in the divisors $\{g_1, g_2, ..., g_s\}$. This algorithm is new, and it is particularly useful for polynomial GCD computations because the gcd G of two polynomials A and B typically has fewer terms, often much fewer, than the quotients A/G and B/G.

We have implemented the division algorithms in the C programming language. We create polynomials in Maple and call our C code from Maple using Maple's foreign function interface (see Ch. 8 of (10)). For comparison we have also implemented Yan's geobucket data structure using dynamic arrays with packed exponent vectors. Details of our geobucket implementation are given in Section 2. In Section 4 we give some benchmarks comparing the simple merging algorithm with Yan's geobucket representation and our heap algorithms, using packed and unpacked exponent vectors.

Our conclusions may be summarized as follows. Simple merging is not competitive with either heaps or geobuckets on sparse problems. The heap algorithms are as fast as geobuckets but use far less memory. Geobuckets do the fewest monomial comparisons, but heaps tend to be faster on large problems because they use cache more efficiently. For all algorithms, packing exponents can significantly improve performance, especially on 64-bit machines.

2 Dynamic Array Implementation

Consider the minimum amount of work that sparse algorithms must do. As noted by Johnson (8), the multiplication fg must construct all $\#f\#g$ products of terms because the monomials generated may be distinct. These terms are merged to form the result. Similarly, to divide f by g we construct the quotient q incrementally while subtracting qg from f. We merge $\#f+\#q(\#g-1)$ terms to do the division. Note, it is $\#g-1$ and not $\#g$ because $-q \times LT(g)$ is constructed to cancel terms so only $-q \times (g - LT(g))$ needs to be merged. Let $r = f - qg$. The number of monomial divisions attempted is $\#q + \#r$. To divide f by $\{g_1, ..., g_s\}$ with quotients $\{q_1, ..., q_s\}$ we merge $\#f + \sum_{i=1}^{s} \#q_i(\#g_i - 1)$ terms and attempt $\sum_{i=1}^{s}(\#q_i)i + (\#r)s$ monomial divisions if for each term we loop through the divisors in order.

Sorting the result imposes an additional cost in monomial comparisons if a function is called to compare terms with respect to an order. The nm terms of a product can be naively sorted using $O(nm \log(nm))$ comparisons, but if the polynomials are sorted we can exploit that fact to do only $O(nm \log(\min(n, m)))$ comparisons. In either case the logarithmic factor is significant – it means that monomial comparisons dominate sparse polynomial computations when the cost of coefficient arithmetic is low.

2.1 Packed Monomial Representations

Following an initial experiment we decided to base our monomial representations on Bachmann and Schönemann's scheme (1), which is used in Singular.

The defining feature of this scheme is that a monomial stores two components: a (possibly weighted) total degree and a vector of exponents. An inline function compares the degree and the exponent vector in lexicographic order, and two global variables invert these comparisons separately. To compare in reverse lexicographic order we reverse the variables and invert all the comparisons. Figure 2.1 shows the unpacked representations of $x^2y^3z^4$ with respect to four common orders with $x > y > z$. Shading is used to indicate where the results of comparisons are inverted.

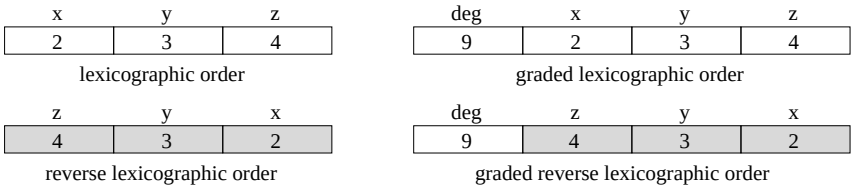


Fig. 1. Unpacked $x^2y^3z^4$ with $x > y > z$.

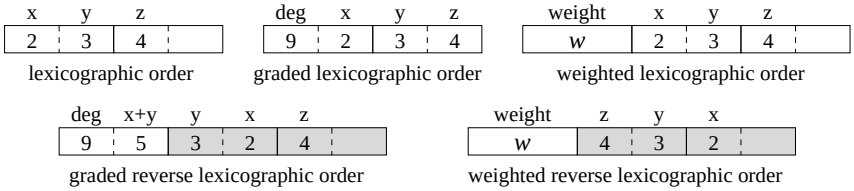


Fig. 2. Packed $x^2y^3z^4$ with $x > y > z$.

To pack monomials we use bitwise or and shift operations on machine words so that byte order is automatically taken into account. Our diagrams use big-endian format. We reserve the most significant bit of each exponent as a guard bit for monomial division. This operation subtracts machine words and uses a bitmask to detect if an exponent is negative. The mask also stores the length of the monomials which is needed by every routine. Weighted orders use the entire first word for the weighted degree since this can be large. We restrict the weights to non-negative integers so that the weighted degree is also a non-negative integer.

For graded orders we use the same number of bits for the total degree as for each exponent so that all monomials up to the maximum degree are encoded efficiently. Note that it is especially easy to determine an optimal packing for these orders using bounds on the total degree. If the polynomials are already sorted with respect to the order then we can examine their leading terms and repack the input in linear time.

Figure 2.1 shows the packed representations of $x^2y^3z^4$ for five monomial orders with two exponents per machine word. Notice how monomial comparisons are reduced to lexicographic and reverse lexicographic comparisons of machine words. The encodings should all be straightforward except for graded reverse

lexicographic order. In that case recall that the total degree only requires as many bits as a single packed exponent. The first word of the monomial, which must be compared lexicographically unlike the rest, would contain relatively little information if it only stored the total degree.

Our first idea was to pack more information into the first word to decide monomial comparisons. Observe that the matrices A and B in Figure 2.1 both describe graded reverse lexicographic order in four variables. Let V be an exponent vector. Then AV is encoded in the first $|V|$ words of the unpacked representation. The matrix B is obtained from A by adding the previous rows of A to each row of A, eliminating all negative entries. Thus BV contains only non-negative integers that are compared lexicographically. We pack as much of BV as possible into the first word of the monomial.

Fig. 3. Matrix representations of graded reverse lexicographic (grevlex) order.

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 4. Packed representations of $x^1y^2z^3t^4u^5v^6w^7$ in grevlex order with 4 exponents per word. *The exponents for w, v, u, and x are redundant. In the second representation, all monomial comparisons can be decided on the basis of the first seven exponents, after looking at two words.*

deg	x..v	x..u	x..t	w	v	u	t	z	y	x	
28	21	15	10	7	6	5	4	3	2	1	

deg	x..v	x..u	x..t	t	z	y	x	w	v	u	
28	21	15	10	4	3	2	1	7	6	5	

However, this does not actually fix the problem since now the second word of the monomial contains information that can be derived from the first. Refer to the top of Figure 4, where $w = 7$, $v = 6$ and $u = 5$ are known from $28 - 21$, $21 - 15$, and $15 - 10$. Thus the second word now provides only one exponent with new information, but we can easily fix this by moving all but the last exponent of the second word to the end of the monomial, as in the bottom of Figure 4. Then for n variables the first n exponents encode all of the information necessary to decide monomial comparisons in grevlex order.

One might wonder why we do not simply encode the vector BV. The reason is that for monomial division one must unpack and decode quotients to check that they are valid. An example is shown below. In fact, we tried this representation initially and found that while it was quite compact for grevlex order, weighted orders were inefficient and reverse lexicographic order could not be implemented.

Eventually we decided to store all of the exponents explicitly, and Bachmann and Schönemann's scheme was the obvious choice.

Example 1. Consider x^2 and y^3 in graded reverse lexicographic order with $x > y > z$. The exponent vectors are $U = [3, 0, 0]$ and $V = [0, 2, 0]$ respectively, and the matrix B is shown below. The difference $BU - BV$ is non-negative even though $U - V = [3, -2, 0]$.

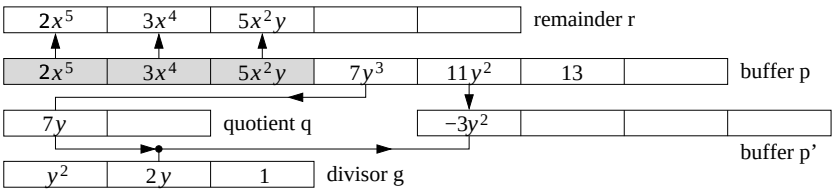
$$B = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \qquad BU = \begin{bmatrix} 3 \\ 3 \\ 3 \end{bmatrix} \qquad BV = \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix} \qquad BU - BV = \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix}$$

2.2 Iterated Merging with Dynamic Arrays

The classical approach to polynomial arithmetic is an iterated merge. To multiply f by g we compute $\sum_{i=1}^{\#f} f_i g$ by adding each $f_i g$ to the previous partial sum using a merge. Similarly, to divide f by g we compute terms of the quotient q while subtracting each $q_i g$ from an intermediate polynomial, which is initially f.

Our first goal was to implement these algorithms while avoiding memory allocation. We use two global arrays or "merge buffers" p and p' which grow dynamically, and all merging takes place from p into p'. If p' does not have sufficient storage to hold the objects being merged then it is enlarged. To amortize this cost we allocate a new p' with 50% more storage than required. To further amortize the cost of memory allocation we reuse p and p' in the next call to an algorithm rather than free them each time.

Fig. 5. Division using two dynamic arrays. *The fourth term of p produced the quotient term y, and we are beginning to merge the rest of p (two terms) with $-y$ times the rest of g (two terms). The buffer p′ is large enough to store the result. Otherwise we would enlarge it to six terms.*



We describe our implementation of the division algorithm. To divide f by g, we copy f into p and increment along the terms of p until we reach the end or we find a term p_i that is divisible by $LT(g)$. We copy the previous terms of p to the remainder and if a reducible term was found, say $p_i = q_j LT(g)$, we merge the rest of p with $-q_j(g - LT(g))$ into p', as shown in Figure 5. The terms of $-q_j(g - LT(g))$ are constructed during the merge. Finally we interchange p

and p' by swapping pointers so that p' becomes p for the next iteration of the algorithm and the storage for p is recycled.

The complexity of this approach was analyzed by Johnson (8). He observed that for a multiplication fg where f has n terms, g has m terms, and fg has nm terms, adding each $f_i g$ to the partial sum can require up to $im-1$ monomial comparisons, making the total number of comparisons $\sum_{i=2}^{n} im - n + 1 \in O(n^2 m)$. A similar result holds for division when the quotient has n terms, the divisor has m terms, and the dividend has nm terms. Thus iterated merging can be very bad when the quotient is large.

It is interesting to note that $O(n^2 m)$ comparisons may be required even if the product or dividend does not have $O(nm)$ terms, if terms introduced by the first $n/2$ summands are canceled by the last $n/2$ summands. We call this an *intermediate blowup in the number of terms*. One unfortunate feature of algorithms that add each $f_i g$ or $q_i g$ to a partial sum is that they allocate storage for all of these terms even when the end result is zero, as it will be for exact division. In Section 3 we will see that the heap algorithms avoid this problem by merging all of the partial products simultaneously.

For sparse polynomials an iterated merge uses about $2s + t$ terms of storage where s is the size of the largest intermediate sum and t is the size of the objects being merged. If we always enlarge the buffers by 50% then we will use storage for about $3s$ terms on average. Quotient(s) and the remainder require additional storage if they are needed.

2.3 Divide and Conquer Merging – Geobuckets

A well-known alternative to iterated merging is divide-and-conquer merging, which is often used for polynomial multiplication. Let f have n terms and let g have m terms. If we compute $\sum_{i=1}^{n} f_i g$ by summing the first $n/2$ and the last $n/2$ summands recursively and adding their sums, then at most $C(n) \leq 2C(n/2) + nm - 1 \in O(nm \log n)$ monomial comparisons are required. The method is efficient because it merges polynomials of similar size.

But how much memory is required? If each recursive call allocates memory for its own result then we can solve the same recurrence to find that $O(nm \log n)$ memory is needed. This is an order of magnitude larger than any possible result. Instead we could reuse a set of geometrically increasing buckets with $\{2m, 4m, \ldots, nm/2\}$ terms for polynomials that we are waiting to merge, plus two arrays with nm and $nm/2$ terms for polynomials that we are currently merging. This simple "geobucket" algorithm is described below.

Geobucket Multiplication.
Input: $f = f_1 + \cdots + f_n$, $g = g_1 + \cdots + g_m$.
Output: fg.

1: Allocate buckets with $\{2m, 4m, \ldots, 2^{\lceil \log_2(n) \rceil - 1} m\}$ terms.
2: Allocate dynamic arrays p and p'.
3: For $i := 1$ while $i \leq n$ do

4: Compute $f_i g$ and store it in p.
5: If $i < n$ merge p and $f_{i+1} g$ into p' and swap p and p'.
6: Set $i := i + 2$.
7: For $j := 1$ while $bucket[j] \neq 0$ do
8: Merge p and $bucket[j]$ into p' and swap p and p'.
9: Set $bucket[j] := 0$ and $j := j + 1$.
10: If $i \leq n$ set $bucket[j] := p$ and $p := 0$.
11: For $j := 1$ to $2^{\lceil \log_2(n) \rceil - 1}$ do
12: If $bucket[j] \neq 0$ merge p and $bucket[j]$ into p' and swap p and p'.
13: Output p.

Thus $f_1 g$ and $f_2 g$ are merged and their sum is stored in bucket 1, then $f_3 g$ and $f_4 g$ are merged and their sum is merged with $f_1 g + f_2 g$ and stored in bucket 2, then $f_5 g$ and $f_6 g$ are merged and their sum is stored in bucket 1, and so on, continuing in the manner of a depth-first search. If $n = 2^k$ it is easy to see that $O(nm)$ storage is used. The buckets contain $(n-2)m$ terms, the array that stores the result will need nm terms, but the other array can have $nm/2$ terms. The total amount of storage required is $2.5nm$ terms – only 50% more than for an iterated merge. If we always grow the arrays by an extra 50% then we can expect to allocate storage for about $3.25nm$ terms in total.

Geobuckets were proposed by Yan (12) with three significant improvements. First, Yan's buckets have a small base and ratio that are independent of any problem to ensure good performance when objects of varying sizes are added to the geobucket. In the algorithm above the base is $2m$ and the ratio is 2, so objects with fewer than m terms could be added more efficiently with a smaller bucket. Second, Yan always tries to store $p + bucket[j]$ in $bucket[j]$ if possible to avoid creating $bucket[j+1]$. This decreases the amount of memory and increases the likelihood of combining terms on dense problems, resulting in fewer monomial comparisons. Finally, Yan describes a reasonably efficient scheme for coalescing the leading terms of the buckets to compute the leading term of the polynomial. This allows us to run the division algorithm with the intermediate polynomial p stored as a geobucket. We state Yan's algorithm below for completeness.

Geobucket Leading Term.
Input: polynomial f stored in $bucket[1 \ldots k]$.
Output: $LT(f)$ or FAIL when $f = 0$, set $bucket[1 \ldots k] := f - LT(f)$.
1: Set $j := 0$, the bucket containing the leading term.
2: For $i := 1$ while $i \leq k$ do
3: If $bucket[i] \neq 0$ and ($j = 0$ or $LM(bucket[i]) > LM(bucket[j])$) set $j := i$
4: else if $bucket[i] \neq 0$ and $LM(bucket[i]) = LM(bucket[j])$
5: Set $LC(bucket[j]) := LC(bucket[j]) + LC(bucket[i])$.
6: Remove $LT(bucket[i])$ from $bucket[i]$.
7: Set $i := i + 1$.
8: If $j = 0$ then $f = 0$ so output FAIL.
9: If $LC(bucket[j]) = 0$ remove this term from $bucket[j]$ and goto step 1.

10: Set $t := LT(bucket[j])$.
11: Remove $LT(bucket[j])$ from $bucket[j]$.
12: Output t.

We implemented Yan's geobuckets using a single dynamic array so that its storage could be reused in subsequent calls. We chose a ratio of two because that is optimal for merging and our smallest bucket (the base) has four terms. We found that geobuckets performed very well, often using fewer monomial comparisons than expected.

For a sparse multiplication producing nm terms geobuckets do $O(nm \log n)$ comparisons and store about $3.6nm$ terms. This number can be derived as follows. The arrays (merge buffers) require nm and $nm/2$ terms, but we will allocate an extra 50% for each. The buckets have nm terms, but the base (two) is independent of m so we expect each bucket to be 75% full. The total is $4nm/3 + (3/2)(nm + nm/2) = (43/12)nm$ terms.

We can make a similar estimate for exact division when the dividend has nm terms, however the complexity is $O(nm \log(nm))$ because of how leading terms are computed. The dividend is placed into the largest bucket, which we expect to be 75% full, so the storage for buckets is $2(4nm/3) = 8nm/3$. Nothing is merged with the largest bucket since $\sum_{i=1}^{\#q} q_i g$ fits entirely in the smaller buckets, so the largest merge that we expect to do is to construct $\sum_{i=1}^{\#q/2} q_i g$ which has $nm/2$ terms. This requires arrays with $nm/2$ and $nm/4$ terms, plus the extra 50% that we allocate, bringing the total number of terms to $8nm/3 + (3/2)(3nm/4) = (91/24)nm$.

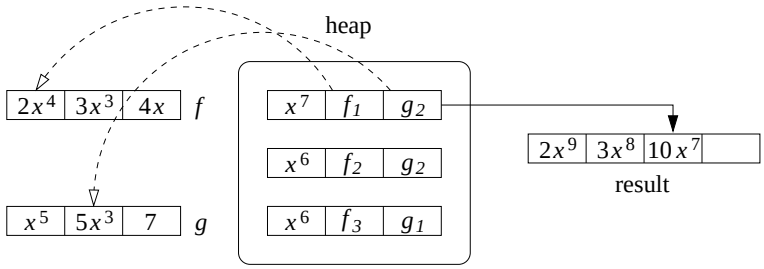
The actual amount of memory that geobuckets need for exact division tends to vary. It can be lower if the leading term computations frequently cancel terms in the buckets, reducing the size of the polynomials that are merged. For random sparse divisions we found that approximately $3.6nm$ terms were used – about the same as for multiplication. The dynamic arrays were often the same size, about $3nm/5$ terms each.

3 Heap Algorithms for Polynomial Arithmetic

The heap algorithms are based on the following idea: rather than merge polynomials one by one into an intermediate object, we do a simultaneous n-ary merge using a heap. Consider the multiplication fg where we merge $f_i g$ for $1 \le i \le \#f$. If we maintain a heap of $\#f$ pointers into g, sorted by the monomial of $f_i g_j$, we can repeatedly extract the largest $f_i g_j$ from the heap, merge it onto the end of the result, and insert its successor $f_i g_{j+1}$ into the heap if $j < \#g$. We illustrate this process in Figure 6 below.

The monomial of $f_i g_j$ is computed and stored in the heap when the term is inserted. It is used to determine the maximum element of the heap. This storage is reused for $f_i g_{j+1}$ so only $O(\#f)$ storage is required, in addition to storage for the result.

Fig. 6. Multiplication of $f = 2x^4 + 3x^3 + 4x$ and $g = x^5 + 5x^3 + 7$ using a heap. *We are extracting* $f_1 g_2 = 10x^7$ *and writing it into the result. The next term* $f_1 g_3 = 14x^4$ *is inserted into the heap, and we extract* $f_2 g_2$ *and* $f_3 g_1$ *to obtain* $15x^6 + 4x^6 = 19x^6$*, the fourth term of the result.*



To divide f by g we merge the dividend f with $-q_i g$ for each term q_i of the quotient. The heap maintains a pointer into f and adds a pointer into each $-q_i g$ as each q_i is constructed. The algorithm extracts the largest term from the heap and continues to extract terms with an equal monomial, adding their coefficients to produce the next term of $f - \sum_{j=1}^{i} q_j g$. If this term is not zero we divide it by $LT(g)$ to obtain either a new term of the quotient q_{i+1}, or the next term of the remainder. When a quotient term is found we insert the *second* term of $-q_{i+1} g$ into the heap, increasing the size of the heap by one, along with the successors of the other terms that were extracted. There is no intermediate blowup in the number of terms that are stored – the maximum number of terms in the heap is $\#q + 1$. We call this a "quotient heap" division.

The heap algorithms above were analyzed by Johnson (8) and used in Altran, one of the first computer algebra systems. For a binary heap of size n, inserting and extracting each term does $O(\log n)$ monomial comparisons. A multiplication that passes nm terms through a heap of size n does $O(nm \log n)$ comparisons – the same as divide-and-conquer. Exact division $f \div g$ with $\#f = nm$, $\#g = m$, and the quotient $\#q = n$, passes $2nm - n$ terms through a heap of size $n + 1$, which is also $O(nm \log n)$ comparisons.

One problem with the heap algorithms is that they do $O(nm \log n)$ comparisons even when the polynomials are dense, whereas the simple merge and the divide-and-conquer algorithms do only $O(nm)$ comparisons. In Section 3.2 we show how to modify the heap to make the heap algorithms efficient in the dense case as well.

Our main contribution is to modify the heap division algorithm to increment along the quotient(s) instead of the divisor(s). The resulting "divisor heap" algorithm does $O(nm \log m)$ comparisons and uses $O(m)$ storage, where m is the size of the divisor(s). Our incentive comes from the gcd problem, where we compute $G = \gcd(A, B)$ and divide A/G and B/G to recover the cofactors. The divisor G is typically small and the quotients (cofactors) are often big. The algorithm

is also useful for computing over small towers of algebraic extensions, when the number of reductions exceeds the size of the extensions.

The modification is easy to do. The algorithm merges f with $-g_i q$ for $2 \leq i \leq \#g$ using a heap of size $\#g$, however we may merge $g_i q_{j-1}$ before q_j is computed, in which case we can not insert the next term $g_i q_j$ into the heap because we can not compute its monomial. However, since $LT(g)q_j > g_i q_j$ for all $i > 1$, we can safely wait for q_j to be computed to insert the terms $g_i q_j$ with $i > 1$ into the heap. We exploit the fact that the term $g_i q_j$ is greater than the term $g_{i+k} q_j$ for $k > 0$, so if $\#q = j - 1$ we encounter the strictly descending sequence $g_2 q_j > g_3 q_j > g_4 q_j > ...$ in order. For each divisor g we store an index s of the largest $g_s q_j$ that is missing from the heap because q_j is unknown. When a new term of the quotient is computed ($\#q = j$) we compute all of the missing terms $\{g_2 q_j, \ldots g_s q_j\}$ and insert them into the heap. Here we give the algorithm for one divisor.

Divisor Heap Division.
Input: $f, g \in F[x_1, ..., x_n], F$ a field, $g \neq 0$.
Output: $q, r \in F[x_1, ...x_n]$ with $f = qg + r$.
1: If $f = 0$ then output $(0, f)$.
2: Initialize $(q, r, s) := (0, 0, \#g)$.
3: Create an empty heap H of size $\#g$ and insert $(-1)f_1$ into H.
4: While the heap H is not empty do
  6: Set $t := 0$.
  7: Repeat
    8: Extract $x := H_{max}$ from the heap and set $t := t - x$.
    9: Case $x = (-1)f_i$ and $i < \#f$: Insert $(-1)f_{i+1}$ into H.
    10: Case $x = g_i q_j$ and $j < \#q$: Insert $g_i q_{j+1}$ into H.
    11: Case $x = g_i q_j$ and $j = \#q$: Set $s := s + 1$ $(s = i)$.
  12: Until H is empty or $LM(t) \neq LM(H_{max})$.
  13: If $t \neq 0$ and $LT(g)|t$ then
    14: Copy $t/LT(g)$ onto the end of q.
    15: For $i = 2, 3, ..., s$ compute $g_i \times (t/LT(g))$ and insert it into H.
    16: Set $s := 1$.
  17: Else if $t \neq 0$ copy t onto the end of r.
18: Output (q, r).

Theorem 1. *The divisor heap algorithm divides f by g producing the quotient q and remainder r using $O((\#f + \#q\#g)log\#g)$ monomial comparisons and using storage for $O(\#g + \#q + \#r)$ terms.*

Proof. We show that at Step 4, $|H| + s - 1 = \#g$ if some $(-1)f_i \in H$ or $|H| + s = \#g$ otherwise. The first time Step 4 is executed, $|H| = 1$, $s = \#g$, and $(-1)f_1$ is in the heap, so the loop invariant holds. Steps 7-11 extract a term from H and either replace it or increment s, unless it was the last term of f. Step 15 inserts $s - 1$ terms into H and sets $s := 1$, maintaining the invariant.

Then $|H| \leq \#g$ since $s \geq 1$. Therefore the storage required is at most $\#g$ terms in the heap plus the terms of q and r. It should be clear that the algorithm

adds terms of f, subtracts terms of each $g_i q$, and uses $LT(g)$ to cancel terms if possible, otherwise moving them to r, so that $f = qg + r$. Since we pass $\#f + \#q(\#g - 1)$ terms through a heap of size $|H| \leq \#g$, the number of monomial comparisons is $O((\#f + \#q\#g) \log \#g)$.

3.1 Heap Optimizations

We present two optimizations that are necessary to reproduce our results. The first is to implement the heap carefully. A surprising number of people are only aware of a bad algorithm for extracting the largest element from a heap, so we present a classical algorithm that is roughly twice as fast on average. As LaMarca and Ladner (9) observe, about 90% of the time is spent extracting elements from the heap so the resulting speedup is almost a factor of two.

We store the heap in a global dynamic array H, with the convention that $H[0]$ is the largest element and the children of $H[i]$ are $H[2i+1]$ and $H[2i+2]$.

```
inline heap_elem heap_extract_max(heap_elem *H, int *n)
{   int i, j, s = --(*n);
    heap_elem x = H[0];
    /* H[0] now empty - promote largest child */
    for (i=0, j=1; j < s; i=j, j=2*j+1) {
        j = (H[j] > H[j+1]) ? j : j+1;
        H[i] = H[j];
    }
    /* H[i] now empty - insert last element into H[i] */
    for (j=(i-1)/2; i>0 && H[s]>H[j]; H[i]=H[j], i=j, j=(j-1)/2);
    H[i] = H[s];
    return x;
}
```

The extraction algorithm promotes the largest child into the empty space at a cost of one comparison per level of the heap H. Then it inserts the last element of the heap into the empty slot, which is a slot for a leaf. However since the last element was already a leaf, we do not expect it to travel very far up the heap. The number of comparisons required is $\log_2(n) + O(1)$ on average.

Compare this with the more commonly known algorithm for shrinking a heap, which moves the last element to the top and, at a cost of two comparisons per level (to find the maximum child and compare with it), sifts it down the heap. Since the last element was already a leaf it is likely to go all the way back down to the bottom, requiring $2\log_2(n)$ comparisons on average.

Our second optimization improves performance when multiple terms are extracted from the heap. It is also necessary to obtain $O(nm)$ comparisons from a chained heap in the totally dense case. We insert and extract batches of terms instead of extracting a term and immediately inserting its successor. This requires a queue to store the extracted terms, however we can partition the heap to store this queue in place, as in heapsort. We insert the successors of all the extracted terms at the end of each iteration. As noted by LaMarca (9), this strategy also produces favorable caching effects.

3.2 Chaining terms with equal monomials

Our next improvement chains heap elements with equal monomials to reduce the number of comparisons. Johnson (8) also experimented with this idea, however our scheme is simpler and we will show that multiplication and division of dense polynomials does $O(nm)$ comparisons.

We chain elements only as they are inserted into the heap, using an additional pointer in the product structure that points to f_i and g_j. In our implementation the pointers to f_i and g_j are not stored in the heap, but in a secondary structure that is accessed only when terms are inserted or extracted. Heap elements store a pointer to this structure and a pointer to the monomial product used for comparisons. The overhead of chaining elements in this way is negligible. The algorithms must be modified to check for chains and extract all the elements of a chain without doing any comparisons.

One final optimization is needed for multiplication. When multiplying fg, we must start with f_1g_1 in the heap and insert each f_ig_1 only after $f_{i-1}g_1$ has been extracted from the heap. Then we have the following two results.

Lemma 1. *Let f and g be dense univariate polynomials with n and m terms, respectively. A heap multiplication fg with chaining does $nm - n - m + 1$ comparisons.*

Proof. We prove a loop invariant: at the beginning of each iteration the heap contains exactly one element or chain. This is true initially since the only element is f_1g_1. Each iteration removes the chain without doing a comparison, producing an empty heap. When we insert the successor terms into the heap all of the monomials are equal because the problem is dense, so all of the terms are chained together at the top of the heap. There are nm terms and $n + m - 1$ unique monomials. The first term with each monomial is inserted for free while the rest use one comparison each to chain. The total number of comparisons is thus $nm - n - m + 1$.

Lemma 2. *Let q and g be dense univariate polynomials with n and m terms and let $f = qg$. Then a quotient heap division $f \div g$ with chaining does $nm - n$ comparisons.*

Proof. We use the same loop invariant: the heap contains exactly one element or chain, which is initially f_1. Each iteration extracts the terms of this chain, adding their coefficients without a comparison, producing an empty heap. If the term is not zero, a new term of the quotient q_i is computed and the monomial of q_ig_1 equal to the monomial of the extracted terms. When we insert its successor q_ig_2 and the successors of all the other terms their monomials are all equal because the problem is dense, and all of the terms are chained together at the top of the heap. If each of the $n + m - 1$ monomials of f is inserted first without any comparisons, the remaining $n(m - 1)$ terms of $-q(g - LT(g))$ will be chained using one comparison each.

Remark: The divisor heap algorithm can also be modified to do nm comparisons in the dense univariate case. Each term q_j of the quotient should insert only g_2q_j if it is not in the heap, and each $g_{i+1}q_j$ should be inserted only after g_iq_j is extracted from the heap. We have not yet implemented this modification.

4 Benchmarks

4.1 The number of monomial comparisons

Our first benchmark (see Table 1 and Table 2) is due to Johnson (8). We multiply and divide sparse univariate polynomials and report the number of comparisons divided by the total number of terms that are merged. Recall that for a sparse multiplication fg this is $(\#f)(\#g)$ and for a sparse division $f = qg$ this is $\#f + \#q(\#g - 1)$. A "structure parameter" S is used to randomly generate polynomials $f = a_0 + a_1x^{e_1} + a_2x^{e_2} + \cdots + a_kx^{e_k}$ with the difference between the exponents satisfying $1 \leq e_{i+1} - e_i \leq S$.

For each problem we generate f and g with n and m terms respectively, multiply $p = fg$ and divide p/g. For multiplication we test both chained and unchained heaps, and for division we test the "quotient heap" and the "divisor heap" algorithms.

Table 1. Multiplication fg and the number of comparisons divided by $(\#f)(\#g)$.

S	$\#(fg)$	$\#f, \#g$	unchained heap	chained heap	geobuckets	direct merge
1	199	100	6.138	.980	1.114	1.475
	1999	1000	9.329	.998	1.027	1.497
10	1025	100	8.339	5.970	2.905	7.239
	10747	1000	11.717	8.478	3.065	8.025
100	5728	100	8.671	8.282	4.690	32.893
	97051	1000	11.879	11.334	5.798	69.191
1000	9364	100	8.805	8.748	5.274	48.073
	566984	1000	11.925	11.852	7.511	324.135

We make a couple of remarks concerning tables 1 and 2. First it should be clear that our implementation of the divisor heap algorithm is not fully optimized. As discussed at the end of Section 3 we should delay inserting products g_iq_j into the heap until after the previous product $g_{i-1}q_j$ is extracted from the heap. This is needed to obtain $O(nm)$ comparisons in the dense case.

Second, it is interesting to see that geobuckets do roughly half the number of comparisons as the heap algorithms in the sparse case, and this ratio improves as the problems become more dense. We tried some improvements to the heap algorithms such as chaining elements while shrinking the heap, however these changes tended to decrease the real world performance of the algorithms.

Table 2. Division $fg \div g$ and the number of comparisons divided by $\#(fg)+\#f(\#g-1)$.

S	$\#(fg)$	$\#f$	$\#g$	quotient heap	divisor heap	geobuckets	direct merge
1	199	100	100	.980	2.627	.980	.980
	1099	100	1000	.989	7.622	.989	.989
	1099	1000	100	.989	1.155	.989	.999
	1999	1000	1000	.998	4.170	.998	.998
10	1025	100	100	5.692	6.480	2.647	4.300
	5856	100	1000	6.493	8.244	2.738	4.872
	5949	1000	100	6.503	7.825	2.748	4.934
	11162	1000	1000	8.646	9.124	2.916	5.473
100	5725	100	100	7.106	7.580	3.945	14.502
	44725	100	1000	7.884	10.594	3.954	19.381
	45358	1000	100	7.696	7.938	4.405	18.231
	96443	1000	1000	10.898	11.438	5.471	42.262
1000	9403	100	100	7.116	7.522	3.992	17.307
	90884	100	1000	7.682	10.608	4.253	23.978
	91141	1000	100	7.658	7.747	4.596	22.736
	571388	1000	1000	10.563	11.056	6.574	142.095

4.2 7 variable cofactor problem

Our next benchmark (see Table 3) simulates a GCD problem. A large sparse polynomial is divided by one of its factors (the GCD) to compute the cofactor. To generate this example we constructed four polynomials $\{f_1, f_2, f_3, f_4\}$ and divided their product $p = f_1 f_2 f_3 f_4$ by f_1, $f_1 f_2$, and $f_1 f_2 f_3$ over $\mathbb{Z}_{32003}$ using graded lexicographic order.

We include the multiplications $(f_1 f_2)(f_3 f_4)$ and $f_1(f_2 f_3 f_4)$ as well. The polynomials all have $\#f_i = 50$ and $\deg(f_i) = 10$. The computations were done on an AMD Opteron 254 2.8 GHz with 8GB of 400MHz RAM and 1 MB of L2 cache.

We report the times and memory overhead with 1, 2, 4, and 8 exponents per 64 bit machine word. For multiplications we subtract the size of the product from the memory totals for the geobucket and merge algorithms, and for divisions we do not include any memory used by the quotient. For the heap algorithms we report the size of the heap and its products. Thus the memory numbers are overhead costs for the various algorithms, not the total memory used. For multiplication we also report the size of the product. For divisions the size of the largest quotient $(f_2 f_3 f_4)$ is at most 8.3 MB.

Table 3. Sparse multiplications and divisions in 7 variables over $\mathbb{Z}_{32003}$ using graded lex order with $\{1, 2, 4, 8\}$ exponents packed into each 64-bit word. $\#f_i = 50$, $\deg(f_i) = 10$, $\#(f_1 f_2) = 2492$, $\#(f_3 f_4) = 2491$, $\#(f_1 f_2 f_3) = 121903$, $\#(f_1 f_2 f_3 f_4) = 4523085$.

$(f_1 f_2) \times (f_3 f_4)$

expon/wd	size of result	chained heap	geobuckets	direct merge
1	310.57 MB	2.630 s (0.38 MB)	7.720 s (994 MB)	332.230 s (371 MB)
2	172.54 MB	1.860 s (0.31 MB)	4.230 s (552 MB)	185.780 s (206 MB)
4	103.52 MB	1.450 s (0.27 MB)	2.550 s (331 MB)	111.960 s (124 MB)
8	69.01 MB	1.240 s (0.25 MB)	1.760 s (221 MB)	75.560 s (83 MB)

$f_1 \times (f_2 f_3 f_4)$

expon/wd	size of result	chained heap	geobuckets	direct merge
1	310.57 MB	1.700 s (0.07 MB)	4.770 s (1143 MB)	8.070 s (483 MB)
2	172.54 MB	1.240 s (0.06 MB)	2.660 s (635 MB)	4.500 s (216 MB)
4	103.52 MB	0.980 s (0.06 MB)	1.690 s (381 MB)	2.800 s (161 MB)
8	69.01 MB	0.880 s (0.06 MB)	1.230 s (254 MB)	1.910 s (107 MB)

$(f_1 f_2 f_3 f_4)/(f_1 f_2 f_3)$

x	quotient heap	divisor heap	geobuckets	direct merge
1	2.000 s (0.13 MB)	8.820 s (18.6 MB)	5.190 s (1793 MB)	7.530 s (944 MB)
2	1.450 s (0.13 MB)	6.570 s (14.9 MB)	2.960 s (996 MB)	4.250 s (524 MB)
4	1.250 s (0.10 MB)	5.270 s (13.0 MB)	1.950 s (598 MB)	2.610 s (315 MB)
8	1.060 s (0.10 MB)	4.530 s (12.1 MB)	1.500 s (398 MB)	1.770 s (210 MB)

$(f_1 f_2 f_3 f_4)/(f_1 f_2)$

x	quotient heap	divisor heap	geobuckets	direct merge
1	3.270 s (0.72 MB)	3.380 s (0.30 MB)	8.020 s (1461 MB)	330.730 s (932 MB)
2	2.290 s (0.65 MB)	2.430 s (0.31 MB)	4.460 s (812 MB)	183.060 s (518 MB)
4	1.840 s (0.62 MB)	1.930 s (0.27 MB)	2.760 s (487 MB)	110.290 s (311 MB)
8	1.520 s (0.60 MB)	1.620 s (0.25 MB)	2.040 s (321 MB)	74.540 s (207 MB)

$(f_1 f_2 f_3 f_4)/f_1$

x	quotient heap	divisor heap	geobuckets	direct merge
1	8.010 s (28.46 MB)	1.990 s (0.07 MB)	8.320 s (1371 MB)	–
2	5.900 s (25.69 MB)	1.480 s (0.06 MB)	4.640 s (762 MB)	–
4	4.750 s (24.29 MB)	1.240 s (0.06 MB)	2.890 s (457 MB)	–
8	3.970 s (23.60 MB)	1.080 s (0.06 MB)	2.210 s (305 MB)	3526.750 s (207 MB)

The heap algorithms perform very well on large examples, despite their higher cost in monomial comparisons. We attribute this to the fact that the algorithms' working memory (the heap of pointers and the monomial products) typically fits in the L2 cache, whereas the RAM of this computer is significantly slower (7x) than the processor.

Also note the effect of packing exponents. The performance of merging and geobuckets is practically linear in the size of the terms, which is 9, 5, 3, or 2 words with a coefficient. The heap algorithms do not benefit as much, but the improvement is certainly noticeable. Going from 64-bit (1 exponent per word) to 16-bit (4 exponents per word) exponents places only modest restrictions on the total degree and improves performance by 40%.

4.3 The effect of fast RAM and a large L2 cache

In our previous benchmark the performance of geobuckets was constrained by the speed of the RAM and the size of the L2 cache in the computer. We know from experience that geobuckets can outperform the heap algorithms under different conditions, because they typically do fewer monomial comparisons.

Our third benchmark (see Table 4) is a smaller problem similar to the previous one. We created three random polynomials $\{f_1, f_2, f_3\}$ and divided their product by f_1 and $f_2 f_3$. This test was run on a 2.4 GHz Intel E6600 Core 2 Duo with 2 GB of 666 MHz RAM and 4 MB of L2 cache using 32-bit words. The RAM is now only 3.6 times slower than the CPU and the number of words in the L2 cache is increased by a factor of eight.

Table 4. Sparse multiplications and divisions in 4 variables over $\mathbb{Z}_{32003}$. Each f_i has degree 30 in each variable. Lexicographic order was used with $\{1, 2, 4\}$ exponents packed per 32-bit word. $\#f_1 = 96$, $\#f_2 = 93$, $\#f_3 = 93$, $\#(f_1 f_2) = 8922$, $\#(f_2 f_3) = 8639$, $\#(f_1 f_2 f_3) = 795357$.

$f_1 \times (f_2 f_3)$

expon/wd	size of result	chained heap	geobuckets	direct merge
1	15.17 MB	0.200 s (0.03 MB)	0.210 s (55.74 MB)	0.650 s (23.21 MB)
2	9.10 MB	0.150 s (0.03 MB)	0.140 s (33.44 MB)	0.470 s (13.92 MB)
4	6.07 MB	0.120 s (0.03 MB)	0.110 s (22.30 MB)	0.360 s (9.28 MB)

$(f_1 f_2 f_3)/(f_1 f_2)$

x	quotient heap	divisor heap	geobuckets	direct merge
1	0.260 s (0.06 MB)	0.460 s (0.55 MB)	0.280 s (70.91 MB)	0.600 s (38.38 MB)
2	0.210 s (0.05 MB)	0.370 s (0.48 MB)	0.220 s (37.38 MB)	0.440 s (27.46 MB)
4	0.170 s (0.05 MB)	0.300 s (0.45 MB)	0.180 s (22.36 MB)	0.350 s (18.30 MB)

$(f_1 f_2 f_3)/f_1$

x	quotient heap	divisor heap	geobuckets	direct merge
1	0.430 s (0.53 MB)	0.280 s (0.03 MB)	0.390 s (55.90 MB)	44.000 s (45.52 MB)
2	0.350 s (0.47 MB)	0.230 s (0.03 MB)	0.300 s (33.54 MB)	28.790 s (27.30 MB)
4	0.280 s (0.43 MB)	0.190 s (0.03 MB)	0.260 s (22.36 MB)	22.150 s (18.20 MB)

The benchmark shows that geobuckets are competitive with the heap algorithms when main memory is fast and a large L2 cache is present. The times above include the cost of allocating memory, and in practice when the geobucket is already allocated it may be faster than a quotient heap, even for sparse problems. Recall that the performance of geobuckets improves on dense problems.

Although geobuckets have a worst case complexity of $O(nm \log(nm))$, in practice they perform as well as a quotient heap. Neither algorithm can compete with a divisor heap when the quotient is large, because its complexity is $O(nm \log m)$.

4.4 Algebraic extensions

Our final benchmark (see Table 5) is a large division with algebraic extensions. We constructed four random polynomials $\{f_1, f_2, f_3, f_4\}$ in $\mathbb{Z}_{32003}[x, y, z, \alpha, \beta, s, t]$ with $\deg(f_i) = 10$ and $LT(f_i) = x^{10}$. We used lexicographic order with $x > y > z > \alpha > \beta > s > t$ with the extensions $\alpha^2 - 3 = 0$ and $\beta^2 + st - 1 = 0$. Thus we are effectively computing with polynomials in $\{x, y, z\}$ with coefficients in $\mathbb{Z}_{32003}[\alpha, \beta, s, t]/\langle \alpha^2 - 3, \beta^2 + st - 1 \rangle$.

We report the times to multiply $(f_1 f_2)(f_3 f_4)$ and $f_4(f_1 f_2 f_3)$ and reduce the product mod $\{\alpha^2 - 3, \beta^2 + st - 1\}$. Then we divide the product by f_1, $(f_1 f_2)$, and $(f_1 f_2 f_3)$ mod $\{\alpha^2 - 3, \beta^2 + st - 1\}$ and reduce the quotients mod $\{\alpha^2 - 3, \beta^2 + st - 1\}$. The divisors in each case are already reduced mod $\{\alpha^2 - 3, \beta^2 + st - 1\}$.

We performed the test on a 3 GHz Intel Xeon 5160 with 16 GB of 666 MHz RAM and 4 MB of L2 cache using 64-bit words. Memory numbers are reported differently because the heap algorithms must store the quotients of $\{\alpha^2 - 3, \beta^2 + st - 1\}$ which are large, whereas geobuckets discard them. Thus we report the total memory allocated by each routine, including reallocations to enlarge the geobuckets and speculative allocations of quotients by the heap algorithms. The heap algorithms allocate quotient(s) using an initial block of 512 terms, doubling this number each time a new block is needed, up to a maximum of 2^{24} terms.

Since the value of packing exponents has already been demonstrated, we packed all seven exponents into one 64-bit word for this test. The results with less packing are consistent with our previous benchmarks.

Table 5. Sparse multiplications and divisions with algebraic extensions. Lexicographic order was used with 7 exponents per 64-bit word. We include the time for the division, the number of monomial comparisons (right), and the total memory allocated. $\#f_1 = 106$, $\#f_2 = 96$, $\#f_3 = 105$, $\#f_4 = 98$, $\#(f_1 f_2) = 8934$, $\#(f_3 f_4) = 8982$, $\#(f_1 f_2 f_3) = 256685$, $\#(f_1 f_2 f_3 f_4) = 1663235$.

	quotient heap	divisor heap	geobuckets
$p = (f_1 f_2)(f_3 f_4)$	11.080 s 9.713×10^8	11.100 s 9.267×10^8	8.510 s 4.218×10^8
reduce product	0.700 s 458.75 MB	0.300 s 166.73 MB	0.610 s 646.54 MB
$p = f_4(f_1 f_2 f_3)$	1.690 s 1.966×10^8	1.680 s 1.546×10^8	2.130 s 8.184×10^7
reduce product	0.670 s 446.07 MB	0.300 s 163.12 MB	0.560 s 642.30 MB
$p/(f_1 f_2 f_3)$	3.060 s 2.862×10^8	11.910 s 6.949×10^8	3.360 s 1.218×10^8
reduce quotient	0.000 s 208.02 MB	0.000 s 64.34 MB	0.000 s 479.98 MB
$p/(f_1 f_2)$	51.430 s 4.097×10^9	35.040 s 2.860×10^9	35.520 s 1.732×10^9
reduce quotient	0.010 s 733.72 MB	0.010 s 81.45 MB	0.010 s 1205.19 MB
p/f_1	49.790 s 2.005×10^9	5.980 s 4.616×10^8	13.140 s 9.100×10^8
reduce quotient	0.190 s 752.61 MB	0.080 s 113.25 MB	0.180 s 1038.96 MB

The divisor heap algorithm clearly shows merit on this example while the quotient heap algorithm does poorly. The reason is simple, $\{\alpha^2 - 3, \beta^2 + st - 1\}$ are small divisors with very large quotients, i.e., they are frequently used to reduce terms during the division. This is typically the case for towers of algebraic extensions. The divisor heap algorithm can also reduce extremely large polynomials of very high degree with respect to a small set of extensions. It scales linearly with the size of the polynomial being reduced and the number of reduction steps required. Its space requirements are also linear in the size of the divisor(s) and their quotient(s).

Geobuckets also perform well on this benchmark. Their overall memory requirements are low because they do not need to store all the quotients and the number of monomial comparisons they do is always competitive. However, performance is not dictated entirely by monomial comparisons. Consider the fourth benchmark $p/(f_1 f_2)$, where geobuckets do half the number of monomial comparisons as a divisor heap only to finish in the same amount of time.

The performance of geobuckets suffers because the amount of memory that they need to access randomly is large, and this decreases the effectiveness of the cache. Imagine what happens when $\beta^2 + st - 1$ is used to reduce one million terms in a row. Geobuckets will merge multiples of this polynomial into the smallest bucket 10^6 times, interspersed with 500,000 merges with the second bucket, 250,000 merges with the third, and so on. When a large bucket is merged it will evict all of the terms from the smaller buckets out of the cache, producing cache misses the next time those bucket are accessed. If the problem is sufficiently large or the L2 cache is small, this will happen frequently.

By contrast, the divisor heap algorithm will do two simultaneous passes over the quotient of $\beta^2 + st - 1$ while randomly accessing a heap with at least 3 elements, a block of at least 2 monomial products, and the terms of the divisor. This is a tiny amount of memory, so almost all of the cache is free to speculatively load terms from the quotient to reduce cache misses. This processor also has an 8-way associative cache, so we expect almost no cache misses from collisions in the cache evicting terms prematurely.

5 Conclusions and Future Work

We have shown how a heap of pointers can be very efficient for sparse polynomial division and multiplication. This performance is primarily due to the very low memory requirements of the algorithms and their cache-friendly design. We have also presented a new algorithm that scales linearly with the size of the quotient(s) by using a heap the size of the divisor(s). This algorithm should have many applications for polynomial computations in the presence of algebraic extensions.

In the future we plan to combine the quotient and divisor heap algorithms to produce an algorithm which is $O(nm \log(\min(n, m)))$, which we believe is optimal. We also plan to implement versions of the heap algorithms that use GMP for large integer arithmetic, and we are experimentally trying to parallelize the heap algorithms as well.

Bibliography

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

[2] W. S. Brown. Altran Users Manual. 4th edition. Bell Labs, Murray Hill, N.J., 1977.

[3] David Cox, John Little, Donal O'Shea. Ideals, Varieties and Algorithms: *An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Springer Verlag, 1992.

[4] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*, Kluwer Academic, 1992.

[5] D. R. Grayson and M. E. Stillman. MACAULAY 2, a software system for research in algebraic geometry. Available at `http://www.math.uiuc.edu/Macaulay2/`

[6] G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 3.0. A Computer Algebra System for Polynomial Computations. Centre for Computer Algebra, University of Kaiserslautern (2005). `http://www.singular.uni-kl.de`.

[7] Richard Jenks, Robert Sutor and Scott Morrison. *AXIOM: The Scientific Computation System* Springer-Verlag, 1992.

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

[9] Anthony LaMarca, Richard Ladner. The Influence of Caches on the Performance of Heaps. *J. Experimental Algorithms* **1**, Article 4, 1996.

[10] M. Monagan, K. Geddes, K. Heal, G. Labahn, S. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 10 Introductory Programming Guide* Maplesoft, ISBN 1-894511-76, 2005.

[11] David Stoutemyer. Which Polynomial Representation is Best? *Proceedings of the 1984 Macsyma Users Conference* Schenectedy, N.Y., (1984) 221–244.

[12] Thomas Yan. The Geobucket Data Structure for Polynomials. *J. Symb. Comput.* **25** (1998) 285–293.