Polynomial multiplication and division using heap.

Michael Monagan and Roman Pearce

Department of Mathematics, Simon Fraser University.

Abstract

We report on new code for sparse multivariate polynomial multiplication and division that we have recently integrated into Maple as part of our MITACS project at Simon Fraser University. Our goal was to try to beat Magma which is widely viewed in the computer algebra community as having state-of-the-art polynomial algebra. Here we give benchmarks comparing our implementation for multiplication and division with the Magma, Maple, Singular, Trip and Pari computer algebra systems. Our algorithms use a binary heap to multiply and divide using very little working memory. Details of our work may be found in [7] and [8].

1 Introduction

This report considers the problem of multiplication and division of *multivariate* polynomials. Polynomial arithmetic is an essential feature of computer algebra systems like Maple. There are two basic representations for multivariate polynomials, the *distributed* representation in which terms are ordered in a monomial ordering and the *recursive* representation. Here is a polynomial in the distributed representation

$$9xy^3z + 4y^3z^2 - 6xy^2z - 8x^3 - 5y^3$$

where the terms are sorted in *graded lexicographical order* with $x > y > z$. In this term ordering, the terms xy^3z and y^3z^2, which both have degree 5, appear before the other terms. Here is the same polynomial in the recursive form

$$-8x^3 \ + \ (9\,zy^3 - 6\,zy^2)x \ + \ (4\,z^2 - 5)y^3.$$

One may think of it as a polynomial in x whose coefficients are polynomials in y whose coefficients are polynomials in z whose coefficients are integers.

In 1984 David Stoutemyer in [9] concluded that, generally, the recursive representation was better for multiplication and division than the distributed representation, and that the recursive dense representation was best overall, even for sparse problems. This was confirmed by Richard Fateman in [3] in 2003 in the following benchmark for multiplying f by g where $f = (1+x+y+z)^{20}$ and $g = f+1$. These timings were obtained on a Pentium III, a 32 bit machine running at 933 MHz. MockMMA is Fateman's own implementation of multiplication using the recursive dense representation in Alegro Common Lisp and the hashing method is Fateman's implementation using a hash table keyed on the monomials.

Our results, however, suggest that the distributed representation can infact be faster than the recursive representation. This we believe is because our codes use algorithms with good asymptotic complexity which use very little working memory and because we pack monomials so that monomial operations are reduced to single machine instructions.

Pari/GP 2.0.17	2.3s	recursive dense array
MockMMA ACL6.1/GMP4.1	3.3s	recursive dense array
Hashing ACL6.1/GMP4.1	4.7s	hash on monomial
Reduce 3.7 (in CSL)	5.0s	sparse recursive linked list
Singular 2.0.3	6.1s	sparse distributed linked list
Macsyma (in ACL 6.1)	6.9s	sparse recursive linked list
Maple VR4	17.9s	sparse distributed array

Table 1: Selected timings (in CPU seconds) from Fateman's 2003 benchmark

2 The Algorithms

Suppose we have two polynomials f and g in $\mathbb{Z}[x_1, ..., x_n]$ which are sorted in the distributed representation with $\#f$ and $\#g$ terms respectively. Consider the problem of computing the product $h = f \times g$ and dividing $h \div f$ to to get the quotient g. If the polynomials are *sparse*, the classical multiplication and division algorithms, which do $O(\#f\#g)$ coefficient operations, may do as many as $O(\#f\#g^2)$ monomial comparisons. An example where this happens is $f = x^n + x^{n-1} + ... + x$ and $g = y^m + y^{m-1} + ... + y$. Maple and Singular solve this problem by using a divide and conquer approach for multiplication and by switching to the recursive representation for division.

In the literature, there is a beautiful idea by Yan in [10] called *geobuckets* which can be used for division as well as multiplication. Singular uses geobuckets for the divisions that occur in Gröbner basis computations. The idea is to represent a polynomial h by a sequence of buckets where the $i'th$ bucket has at most 2^i terms of h. When we add (subtract) a polynomial with k terms from h we add (subtract) it from the bucket j with $2^{j-1} < k \leq 2^j$ terms using merging. If the result is too large to store in bucket j then the result is added to the next bucket using merging, and so on, until it fits. The total number of monomial comparisons to compute $h = f \times g$ and $g = h/f$ becomes $O(\#f\#g \log(\#f\#g))$.

But it as an older idea of Johnson in [6] from 1974 that we have found is superior on our modern computers with their many levels of cache. The idea is to use an auxiliary data structure, a binary heap, sorted on the monomials in the product h for multiplying $f \times g$. Johnson's multiplication algorithm does $O(\#f\#g \log \min(\#f, \#g))$ comparisons and requires $O(\min(\#f, \#g))$ storage for the heap. Johnson implemented the algorithm in the ALTRAN system but it seems to have been forgotten. We have also observed that since the terms in the product h are generated in order, the coefficient arithmetic can be done in temporary storage in such a way that the entire multiplication $f \times g$ generates no garbage. This is a clear advantage. In [7] we found that our heap implementation beat our geobucket implementation for sparse polynomials.

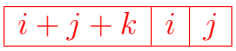
Johnson also used a heap to test if f divides h exactly, with zero remainder, and to compute the quotient g using $O(\#h + \#f\#g \log \#g)$ comparisons. This is good if the number of terms in the quotient g is smaller than that of the divisor f which is the case in some contexts but not others. In [7] we have devised a new algorithm to (i) get the complexity down to $O(\#h + \#f\#g \log \min(\#f, \#g))$ comparisons, i.e., the same as multiplication, (ii) extend the algorithm to compute the remainder r efficiently by using pseudo-division to avoid

arithmetic with fractions and (iii) do everything without creating any garbage.

Dividing the total cost by $(\#f\#g)$, we get the cost per term in the multiplication and division algorithms is cost of the coefficient arithmetic plus the cost of $O(\log \min(\#f, \#g))$ monomial comparisons. One of the reasons why the distributed representation is slow compared with the recursive representation is that we must compare monomials. In general this requires a function call and a loop over the degrees in each variable. One way to reduce this cost is to pack multiple exponents into one machine word.

2.1 Monomial Representations

In [1], Bachmann, Schönemann describe various packing schemes and show that this is helpful. It is most helpful, obviously, if we can pack the entire monomial into one machine word. Consider the monomial $x^i y^j z^k$ in graded lex order with $x > y > z$. If we pack this in one 64 bit word



i.e., with the total degree $i + j + k$ in the leading 22 bits, i in the next 21 bits, and j in the bottom 21, then we can compare two monomials X and Y by doing a single unsigned machine integer comparison and we can multiply XY by doing an unsigned machine integer addition. Thus if the number of variables and their degrees is not too large, then we essentially eliminate the cost of the monomial arithmetic altogether. On a 64 bit machine, we can pack a polynomial in 8 variables with degree up to 255. This encompasses essentially all polynomials in 8 variables arising in practice. The 64 bit word makes this very effective.

Note, the reason we prefer graded lexicographical order over pure lexicographical order is so that we can do a polynomial division without having to check for exponent overflow. For if one uses lexicographical order, and the remainder is not zero, the degree of the remainder in the non-main variables can be larger then the degree of the non-main variables in the inputs. One needs an overflow detection bit to detect overflow if we use a packed representation. The good thing about graded orderings is that the degrees remain bounded and hence no overflow detection is needed.

2.2 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 (SDMP) is a C library that uses heaps to compute with sparse polynomials. 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 .

We have included timings for the computer algebra systems Pari, Magma (see [2]), Maple 11, Singular (see [5]) and Trip (see [4]). Magma was designed for computational algebra and group theory, Maple is general purpose, and Singular was designed for algebraic geometry. They use the distributed representation. Pari was designed primarilly for computational number theory and Trip, a new system, is designed for celestial mechanics. Pari and Trip use the recursive representation.

2.3 Fateman's Benchmark

Our first benchmark, a dense problem, is due to Fateman [3]. Let $f = (1 + x + y + z + t)^{30}$ and $g = f + 1$. We multiply $h = f \times g$ and divide $q = h/f = g$. The polynomials f and g have 46,376 terms and 61 bit coefficients. The product h has 635,376 terms and 128 bit coefficients. We use graded lexicographic order with $x > y > z > t$. That our code beat's Pari which is using the recursive dense representation is a surprise.

Table 2: Dense multiplication and division over $\mathbb{Z}$

$46376 \times 46376 = 635376$	$h = f \times g$		$q = h/f$	
	time	space	time	space
LOWER BOUND	15.50s		15.50s	
SDMP (1 word monomial)	47.42s	19.8mb	68.16s	3.4mb
SDMP (4 word monomial)	107.43s	45.2mb	126.32s	5.1mb
Trip v0.99 (rationals)	108.22s	68.7mb	NA	
Pari/GP 2.3.3	512.18s	–	283.44s	–
Magma V2.14-7	679.07s	160.4mb	610.62s	70.7mb
Singular $3-0-4$	1482.36s	98.1mb	364.49s	80.5mb
Maple 11	15986.17s	–	13039.25s	–

That the Singular timing for division is three times faster than multiplication is because Singular automatically switches to using the recursive representation for division (but not for multiplication).

The time of 15.50 seconds represents a lower bound on the time for multiplying $f \times g$ and dividing $h \div f$ on this particular machine. We obtain this time by writing down the coefficients of f and g in an array and viewing them as dense univariate polynomials and multiplying them using assembler to do the coefficient arithmetic. That is, we are bounding from below the cost of the coefficient arithmetic. Thus we are within a factor of 3 to 4 from what is possible.

2.4 Sparse Problems in Many Variables

Our second benchmark is a sparse computation in ten 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 $h = f \times g$ and divide $q = h/f$. We use $n = 10$ and $d = 5$. Then f has 26,599 terms, g has 36,365 terms, and h has 19,631,157 terms. We will use lexicographic order with $x_1 > x_2 > \cdots > x_{10}$.

The Magma timing for division is very slow compared with multiplication. This is because this benchmark is sparse (the first is dense) and whereas Magma is using a hash-table based algorithm for multiplication, it is using the classical algorithm for division.

Notice the space efficiency of our division algorithm. It needs space for the output (the quotient q), the heap and temporary registers. This is much smaller than the size of the dividend h.

Table 3: Sparse multiplication and division over $\mathbb{Z}$

$26599 \times 36365 = 19631157$	$h = f \times g$		$q = h/f$	
	time	space	time	space
SDMP (1 word monomial)	37.14s	303mb	41.33s	3.4mb
SDMP (10 word monomial)	174.87s	1,667mb	162.37s	14.4mb
Pari/GP 2.3.3	109.27s	–	109.69s	–
Trip v0.99 (rationals)	221.91s	2,123mb	NA	
Magma V2.14-7	313.02s	2,365mb	5744.60s	1,753mb
Singular 3-0-4	655.25s	1,538mb	206.60s	1,390mb
Maple	14053.37s	–	10760.36s	

References

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

[2] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I: The user language. *J. Symbolic Comput.*, **24**(3-4):235-265, 1997.
Magma homepage: `http://magma.maths.usyd.edu.au/magma`

[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.
Trip homepage: `http://www.imcce.fr/Equipes/ASD/trip`

[5] 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).
Singular homepage: `http://www.singular.uni-kl.de`

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

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

[8] M. Monagan, R. Pearce. Sparse Polynomial Pseudo Division using a Heap. In preparation for the special issue "Milestones in Computer Algebra" of *J. Symb. Comput.* in honor of Keith Geddes' 60th birthday.

[9] D. Stoutemyer. Which Polynomial Representation is Best? Surprises Abound! *Proceedings of the 1984 Macsyma Users' Conference* Schenectady, New York, pp. 221–243, 1984.

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