POLY : A new polynomial data structure for Maple 17 [⋆]

Michael Monagan and Roman Pearce

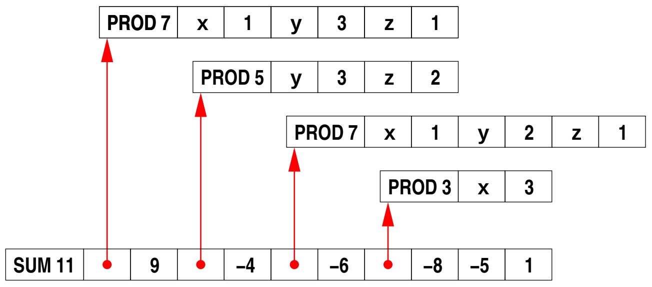
Department of Mathematics, Simon Fraser University,
Burnaby, British Columbia, V5A 1S6, Canada
`mmonagan@cecm.sfu.ca` and `rpearcea@cecm.sfu.ca`

Abstract. We demonstrate how a new data structure for sparse distributed polynomials in the Maple kernel significantly accelerates several key Maple library routines. The POLY data structure and its associated kernel operations (degree, coeff, subs, has, diff, eval, ...) are programmed for compactness, scalability, and low overhead. This allows polynomials to have tens of millions of terms, increases parallel speedup, and improves the performance of Maple library routines.

1 Introduction

Figure 1 below shows the default polynomial data structure in Maple 16 and all previous versions, for the polynomial $f = 9xy^3z - 4y^3z^2 - 6xy^2z - 8x^3 - 5$. It is a "sum-of-products" where each term has a separate Maple object, a `PROD`, to represent the monomial. To compute the degree of f, a coefficient in x, test for a subexpression, or do almost anything else, the Maple kernel must recursively descend through multiple levels of dags. This involves extensive branching and random memory access, which prevents Maple from achieving high-performance on modern computer processors.

Fig. 1. Maple's sum-of-products representation encodes each object in an array of words where the first word encodes the type and length (in words) of the object.



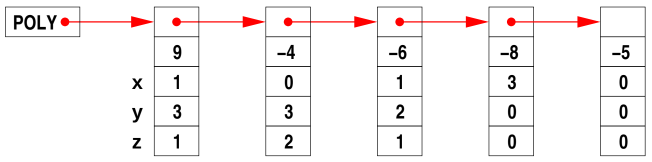
⋆ This work was supported by Maplesoft and the MITACS NCE of Canada.

For example, to compute the degree of f in x, when Maple sees x in the monomial $\begin{array}{|c|c|c|c|c|c|c|}\hline \texttt{PROD} & x & 1 & y & 3 & z & 1 \\ \hline \end{array}$, because Maple does not have a dedicated polynomial data structure, Maple does not know that the next factor does not have x in it. This "monomial" could be $x(x+y)^3z$ or $x\sin(x)^3z$. Thus, computing the degree of a polynomial with t terms in n variables is $O(nt)$ in Maple.

Monomial operations are particularly slow. To multiply xy^3z by xz Maple forms the product $\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|}\hline \texttt{PROD}\ 11 & x & 1 & y & 3 & z & 1 & x & 1 & z & 1 \\ \hline \end{array}$. It then "simplifies" this product to obtain $\begin{array}{|c|c|c|c|c|c|c|}\hline \texttt{PROD}\ 7 & x & 2 & y & 3 & z & 2 \\ \hline \end{array}$ Note, since the variables in a `PROD` are not sorted, Maple cannot simply merge the two monomials on the variables. Maple searches the `PROD` on the variables before adding exponents of like variables. Finally, because Maple stores unique copies of objects, the resulting `PROD` object is hashed and inserted in an internal table if it not already present. In all, there are many function calls and many loops. We estimate that Maple takes more than 200 clock cycles for each monomial multiplication involving 3 variables.

For comparison, below is Singular's data structure for the same polynomial. Singular is representative of several computer algebra systems that use a linked list of terms and dense exponent vectors to represent monomials. The exponent vector makes computing the degree of a polynomial $O(t)$ instead of $O(nt)$. Multiplication of monomials consists of an allocation and a loop of fixed length to add exponents. The linked list allows terms to be merged in place. For thest reasons, Singular is much faster than Maple for polynomial multiplication.

Fig. 2. Singular's polynomial representation uses a linked list of terms



Singular is representative of several computer algebra systems which have a dedicated distributed polynomial representation with obvious advantages. Monomial multiplication in Singular requires a single allocation of memory and a single loop to add exponents. This is the main reason Singular is faster than Maple at polynomial multiplication.

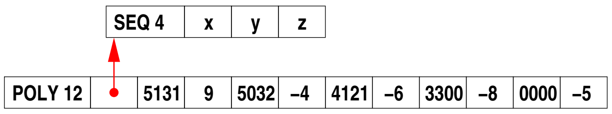
The relatively slow performance of Maple for polynomials lead us to develop a high performance C library for sparse polynomial arithmetic which we integrated into Maple in [16]. To multiply two polynomials, we first convert the Maple inputs to a special polynomial representation, then we call our library to compute the product using our parallel algorithm from [18], then we convert the result to Maple's sum-of-products representation. This substantially improved Maple's performance for large polynomials. However, for very sparse polynomials, and small polynomals, the overhead of constructing the sum-of-products representation was a bottleneck. That overhead often negated the parallel speedup achieved by our library. Furthermore, algorithms in the Maple library continued to use the sum-of-products structure, so the cost of ancillary operations like computing the degree or coefficients in a variable could easily overshadow the cost of arithmetic. These problems are addressed by the introduction of a new polynomial data structure in the Mapler kernel described here. This new data structure is now in Maple 17 which was released in April 2013.

Our paper is organized as follows. In Section 2 we describe the new POLY data structure and list some of its obvious advantages. We give details about how we have integrated it into Maple. In Section 3 we describe how we implemented various Maple kernel operations and give benchmarks demonstrating the improvement in their performance. In Section 4 we consider the impact of the POLY data structure and improved kernel routines on Maple library codes and measure improved parallel speedup in polynomial arithmetic. We end with a conclusion.

2 The POLY Data Structure

Figure 3 below shows our new data structure for sparse distributed polynomials. The first word is a header word, which notes the length and type of the object. The second word points to the variables, which are sorted in Maple's canonical ordering for sets. This is followed by the monomials and coefficients, where the monomials encode the exponents and the total degree as a single machine word. E.g., for xy^2z^3 we store the values $(6, 1, 2, 3)$ as $6 \cdot 2^{48} + 2^{32} + 2 \cdot 2^{16} + 3$, using 16 bits each on a 64-bit machine. Terms are sorted into graded lexicographical order by comparing the monomials as unsigned integers. This gives a canonical representation for the polynomial. Small integer coefficients $-2^{62} < x < 2^{62}$ are encoded as $2x + 1$, so that the rightmost bit is 1. Larger integers are a pointer (with rightmost bit 0) to a GMP multiprecision integer [8]. The current implementation requires all coefficients to be integers.

Fig. 3. The new packed representation for $f = 9xy^3z - 4y^3z^2 - 6xy^2z - 8x^3 - 5$.



Five advantages of the new representation are readily apparent.

1. It is much more compact. Polynomials use two words per term instead of $2n + 3$ words, where n is the number of variables. For polynomials in three variables this saves over a factor of four.
2. Monomial comparisons become machine word comparisons and monomial multiplication becomes machine word addition (provided there is no overflow), and monomial division becomes subtraction with a bitwise test for failure. This dramatically speeds up polynomial arithmetic.

3. Explicitly storing variables and sorting the terms lets us perform many common Maple idioms without looking at all of the terms, e.g. `degree`(f) (total degree), `indets`(f) (extract the set of variables), `has`(f, x), and `type`($f, polynom$).
4. Other operations such as `degree`(f, x), `diff`(f, x), or `coeff`(f, x, i) (extract the coefficient of x^i) access memory sequentially to make good use of cache. We can isolate groups of exponents using masks. This eliminates branching and loops at the level of the exponents.
5. For large polynomials, we avoid creating many small Maple objects (the `PROD`s) that must be simplified by Maple's internal simplifier and stored in Maple's `simpl` table, an internal hash table of all Maple objects. They fill the `simpl` table and slow down Maple's garbage collector.

The idea of packing monomials in one or more computer words is not new; the ALTRAN computer algebra system [11] allowed the user to pack monomials in lexicographical order to conserve memory. Bruno Buchberger [4] experimented with packed monomials using the gradex lexicogrpahical orering when he was computing Gröbner bases in the early 1960s because of the scarcity of main memory. A number of special purpose computer algebra systems have supported monomial packing. For example, the McCaulay2 computer algebra system [9] for computing Gröbner bases supports packing for the lexicographical and graded reverse lexicographical monomial orderings and the Magma computer algebra system [3] dynamically packs monomials using bytes. In [1], Bachmann and Schönemann compared the graded packing with packings for other monomial orderings for Gröbner basis computation.

We have integrated the POLY data structure into the Maple kernel and it is used by default as of Maple 17. When Maple parses input from the user or a file it creates the sum of products structure which is then simplified. When the simplifier detects an expanded polynomial with integer coefficients and at least two terms it creates a POLY structure when: (i) the number variables $n < \beta/2$ for β-bit machines, (ii) the total degree d satisfies $1 < d < 2^b$ where $b = \lfloor \beta/(n+1) \rfloor$, and (iii) the variables are Maple *names* with regular evaluation rules, e.g. $\{x, y_1, \pi\}$ but not *infinity* or *undefined*. Otherwise the sum-of-products format is retained. On output, and for cases where it was convenient or necessary, Maple recreates a sum-of-products structure for a POLY and caches it. Thus Maple 17 uses two representations for polynomials, automatically selecting between the two. All conversions between representations are automatic and invisible to the Maple user.

Note, the monomial encoding is determined solely by the number of variables in the polynomial. We use b bits for the exponent in each variable and for the total degree. This means that aritmetic operations between polynomials in the same variables do not require repacking.

Note, for polynomials with total degree $d = 1$, we chose not to store them in as a POLY dag because Maple's sum-of-products representation is better in this case. For example $f = 2x + 3y + 4z + 5$ is represented as $\boxed{\texttt{SUM 9}}\boxed{2}\boxed{x}\boxed{3}\boxed{y}\boxed{4}\boxed{z}\boxed{5}\boxed{1}$. This is compact and monomials are not explicitly represented.

We chose the graded lex ordering rather than pure lexicographical ordering for several reasons. First, the graded ordering appears more natural for output. Second, when multiplying $a \times b$, if the total degree $d = \deg a + \deg b$ does not overflow, that is, $d < 2^b$, then the entire product can be computed without overflow and with no overflow detection. This allows us to look at only the leading terms of polynomials and predict overflow in $O(1)$ time. Third, in the division algorithm, if one uses pure lexicographical order, degrees in the remainder can increase dramatically and overflow the exponents. For example, consider the following division in lexicographical order with $x > y$.

$$x^2y^5 + y^3 \div x^2y + xy^5$$

The quotient is y^4 and the remainder is $-xy^9 + y^3$. If we had 3 bits per variable, the y^9 would overflow. In contrast, when a graded ordering is used the total degree of the monomials in the division algorithm always decreases. In our example the leading term of the divisor would be xy^5 and the division

$$x^2y^5 + y^3 \div xy^5 + x^2y$$

would result in the quotient x and remainder $-x^3y + y^3$.

We encode the monomial $x^iy^jz^k$ as $\boxed{d\,i\,j\,k}$ where $d = i + j + k$. Notice, however, that we could store $\boxed{d\,i\,j}$ and recover $k = d - i - j$ as needed to allow for more bits. We rejected this idea because it adds too much code complexity, both in the kernel routines and in external libraries that must support POLY. It also makes the cost of some monomial operations $O(n)$ instead of $O(1)$.

We expect to pack many practical problems into 64-bit words, which is the norm for today's computers. For example, if a polynomial has 8 variables then we store 9 integers for each monomial using $\lfloor 64/9 \rfloor = 7$ bits each. So the POLY dag can accommodate polynomials in 8 variables of total degree up to 127. Table 1 below shows the number of bits per exponent for a polynomial in n variables assuming a 64-bit word. Column lex shows how many bits would be available if lexicographical order were used and one bit (at the top) were left unset to test for overflow. To check for overflow in $a+b$ we would compute $(a+b) \oplus a \oplus b$. This sets the bits that are carried into (see [21]) after which we can apply a mask.

Table 1. The number of bits per exponent for n variables in the two orderings.

n	lex	grlex	unused	n	lex	grlex	unused	n	lex	grlex	unused	n	lex	grlex	unused
1	63	62	2	7	9	8	0	13	4	4	8	19	3	3	4
2	31	21	1	8	7	7	1	14	4	4	4	20	3	3	1
3	21	16	0	9	7	6	4	15	4	4	0	21	3	2	20
4	15	12	4	10	6	5	9	16	3	3	13	22	2	2	18
5	12	10	4	11	5	5	4	17	3	3	10	...	...	...	...
6	10	9	1	12	5	4	12	18	3	3	7	31	2	2	0

Column grlex is our packing. It shows how many bits are available for each variable and for the total degree. For univariate polynomials we do not store the degree because it simply duplicates the exponent. Instead we restrict the degree to the range of Maple immediate integers (62 bits signed x is stored as $2x + 1$) to avoid handling multiprecision exponents in conversions. The table also shows the number of unused bits at the top of each word.

When this work was presented at the Asian Symposium on Computer Mathematics in Beijing in October 2012, Joris van der Hoven asked "Why don't you use the extra bits for increased total degree?". For example, in Table 1 we see that for $n = 10$ variables, we allocate $b = 5$ bits per exponent and 5 bits for the total degree leaving 9 unused bits. The answer we gave was that using those bits for the total degree would increase the complexity of the implementation significantly. In the following year, when computing deteriminants of various matrices of polynomials, we encountered three determinants which do not fit in POLY. However, in all three cases, they would fit if we used the unused bits for total degree. For example, consider the $n \times n$ Vandermonde matrix V_n where the (i, j)'th entry of V_n is x_i^{j-1}. Shown below is V_4

$$V_4 = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \\ 1 & x_4 & x_4^2 & x_4^3 \end{bmatrix}$$

The determinant D_n of V_n is a polynomial in n variables of total degree $0 + 1 + 2 + 3 + ... + n - 1 = n(n - 1)/2$. Looking at Table 1, we can see that D_{10} which as total degree 45 does not fit in POLY because there are only 5 bits for the total degree. However, it would easily fit if the extra 9 bits were used for the total degree. In fact, D_n it would fit for $n = 11, 12, 13$ and 14 as well. Because of this and the significant number of unused bits for $9 \le n \le 14$ variables, we have decided to use them for the total degree and are presently implementing this.

3 Operations in the Maple Kernel

The new representation allowed us to design new high performance algorithms for the Maple kernel. In the old data structure most operations are $O(nt)$ where n is the number of variables and t is the number of terms. Maple must examine the entire sum of products structure because its contents are unknown.

With the new data structure we can often avoid doing expensive operations on all of the terms, or we can do them much more efficiently. Our first example is the command `diff`(f, x). To compute the derivative with respect to x in the sum of products representation, Maple searches for x in each `PROD`. If x is found it copies the `PROD`, decreases the exponent of x, and scales the coefficient.

Our algorithm for `diff`(f, x) first locates x in the set of variables. If x is not there it returns zero. Otherwise let x be variable k of n, let $s = \lfloor 64/(n + 1) \rfloor$ be the width of the exponents, $sk = s(n - k)$ is the shift to find the exponent of x, and $b = 2^s - 1$ is a mask of s bits. The core of `diff`(f, x) is a loop that updates the exponent of x and the total degree (if present) using one subtraction.

```
/* subtracted from monomial */
d = 1 << sk;
if (n > 1) d += 1 << (s*n);
for (i=j=2; i < LENGTH(f); i+=2) {
        m = f[i];            /* monomial */
        e = (m >> sk) & b;   /* exponent */
        if (!e) continue;    /* skip the constant */
        r[j+0] = m - d;      /* new monomial */
        r[j+1] = mulint(f[i+1], IMMEDIATE(e), NULL);
        j += 2; }
```

The new monomials in the derivative remain sorted in the graded ordering. For example, consider $f = 9xy^3z - 4y^3z^2 - 6xy^2z - 8x^3 - 5$. If we differentiate with respect to x we obtain $f' = 9y^3z + 0 - 6y^2z - 24x^2 + 0$ and the non-zero terms remain sorted in the graded monomial ordering. Thus we can differentiate in $O(n + t)$ instead of $O(nt)$.

As a second example, consider $\texttt{degree}(f, \{x, z\})$; for $f \in \mathbb{Z}[x, y, z]$. We exploit a trick from Hacker's Delight [21]. If a and b are packed monomials $(a-b) \oplus a \oplus b$ sets the bits that are borrowed to subtract $a-b$. The absence of underflow in the exponent fields indicates that two monomials divide and $a - b$ is their quotient. To test for division in $\{x, z\}$ only, we construct a mask m with 1 bits above the x and z fields and compute `((a-b) ^ a ^ b) & m`. If this is zero we know b divides a in $\{x, z\}$, so $\texttt{degree}(b, \{x, z\}) < \texttt{degree}(a, \{x, z\})$ and we can skip over b without computing its degree. This optimization is also used for `lcoeff` which computes the leading terms of polynomials. In the case of `degree` we can stop when the total degree of the monomials is less than or equal to the largest degee already found.

Another optimization that we use is binary search. Consider $\texttt{has}(f, x^5)$ and $\texttt{coeff}(f, x, 5)$. In both cases it is pointless to examine terms of total degree less than five, but rather than test each monomial with an additional branch in the main loop, we use binary search to adjust the endpoint for linear search.

Our results are summarized in Table 2 which reports the speedup for kernel operations on a polynomial with 990,000 terms in 3 variables, constructed from

$$\texttt{f} := \texttt{expand}(\texttt{mul}(\texttt{randpoly}(\texttt{i}, \texttt{degree} = 100, \texttt{dense}), \texttt{i} = [\texttt{x}, \texttt{y}, \texttt{z}])) :$$

Note that none of these operations are parallelized, and the cost for evaluation is added to the other commands if you are using Maple interactively.

Some algorithms we need to sort polynomials after modifying the monomial data. We sort by treating the monomials as unsigned 64-bit integers and using an inplace MSD American flag radix sort [15]. However most operations do not require any sorting.

The `coeffs(f,x)`, `eval(f,x=6)` and `taylor(f,x,n)` commands all need the coefficients of f in x. Suppose f is a polynomial in $\{w, x, y\}$. For each monomial $w^i x^j y^k$ in f, encoded as $\boxed{\texttt{dijk}}$ where $d = i + j + k$ is its degree, with a constant number of masks and bit operations (seven suffices) we move j, the degree of x to the front to obtain the $\boxed{\texttt{jdik}}$. We sort this modified data to group terms by power of x with ties broken by the monomial ordering on $\{w, y\}$. From that we extract the coefficients in x, already sorted, in linear time.

Table 2. Improvements for Maple kernel operations.

command	description	Maple 16	Maple 17	speedup
`f;`	top level evaluation	0.162 s	0.000 s	$\to O(n)$
	evaluate the variables, interactive statements incur this cost			
`coeff`$(f, x, 20)$	coefficient of x^{20}	2.140 s	0.005 s	420x
	linear search to degree 19, binary search for univariate f			
`coeffs`(f, x)	extract all coefficients in x	0.979 s	0.119 s	8x
	reorder exponents $[d, x, y, z] \to [x, d, y, z]$, sort to collect in x			
`frontend`$(g, [f])$	subs functions for variables	3.730 s	0.000 s	$\to O(n)$
	there's nothing to do since POLY has no functions			
`degree`(f)	total degree in all variables	0.175 s	0.000 s	$\to O(1)$
`degree`(f, x)	degree in x	0.073 s	0.003 s	24x
`diff`(f, x)	differentiate wrt x	0.956 s	0.031 s	30x
`eval`$(f, x = 6)$	compute $f(6, y, z)$	3.760 s	0.175 s	21x
	reorder exponents $[d, x, y, z] \to [d, y, z, x]$, sort, Horner form			
`expand`$(2xf)$	multiply by a term	1.190 s	0.066 s	18x
`has`(f, x^{101})	search for subexpression	0.040 s	0.002 s	20x
	linear search to degree 100, $O(n)$ for names, $O(\log t)$ for terms			
`indets`(f)	set of indeterminates	0.060 s	0.000 s	$\to O(1)$
`lcoeff`(f, x)	leading coefficient in x	0.058 s	0.005 s	11x
	monomial division avoids comparisons, stop early using degree			
`op`(f)	extract terms of f	0.634 s	2.420 s	0.26x
	create old structure, cached until the next garbage collection			
`subs`$(x = y, f)$	replace variable	1.160 s	0.076 s	15x
	add exponents of like variables, sort and combine like terms			
`taylor`$(f, x, 50)$	Taylor series to $O(x^{50})$	0.668 s	0.055 s	12x
	reorder exponents $[d, x, y, z] \to [x, d, y, z]$, sort to collect in x			
`type`$(f, \mathit{polynom})$	type check	0.029 s	0.000 s	$\to O(n)$
	type check the variables, other types are optimized as well			

The biggest improvement we have seen for a Maple library command is the `collect` command, which is used to write a polynomial in recursive form. For example, if $f = xy^3 + x^2y - x^2z + xyz - 2$ then `collect(f,x)` will rewrite f as $(y - z)x^2 + (y^3 + yz)x - 2$. The Maple code for the `collect` command uses the `series(f,x,3)` command to implement this. Since the `series` command is 8x faster, we did not expect collect to be $6.616/0.123 = 54$ times faster. Below is a profile showing that most of the time in Maple 16 was not in `series` at all, but rather in the `frontend` and `indets` commands whose costs are now negligible.

In our new Maple, the cost of `frontend` and `indets` are now negligible since they no longer need to descend into the sum-of-products dag. In the new Maple, they only need to look at the variables which costs $O(n)$. Why were `frontend` and `indets` so expensive? They need to search the sum-of-products dag to see if there are any indeterminates which are not variables. They are looking for objects

Table 3. Profiles for executing `collect(f,x)` in Maple 16 and Maple 17.

			Maple 16		Maple 17	
function	depth	calls	time	time%	time	time%
frontend	1	1	3.932	59.43	0.000	0.00
indets	1	2	1.522	23.00	0.000	0.00
series	1	1	0.919	13.89	0.109	88.62
collect/recursive	1	1	0.160	2.42	0.010	8.13
collect/series	1	1	0.083	1.25	0.004	3.25
collect	1	1	0.000	0.00	0.000	0.00
total:	6	7	6.616	100.00	0.123	100.00

like $x^{1/2}$, $\sin(x)$, 2^n, etc. But our polynomial has none; it only has variables x, y and z in it. In order to do this they pick apart each product and each power, e.g., given x^3yz^4 they recursively constructs x^3 and z^4 as new objects before descending them to see $x, 3, z, 4$.

3.1 Unpacking

The one case where we lose is when we must unpack the POLY dag and convert to the old data structure. The Maple command `op(f)` constructs a sequence of all the terms of f. Other Maple commands which effectively do the same thing include the common Maple programming idioms:

`map(g,f)`	apply the function g to each term of f
`for t in f do`	iterate over the terms of f
`indets(f,t)`	extract all subexpressions in f of type t

Each term, e.g. $8xy^2$, is stored as $\boxed{\texttt{SUM}}\boxed{\uparrow P}\boxed{8}$ where P is the monomial stored as $\boxed{\texttt{PROD}}\boxed{x}\boxed{1}\boxed{y}\boxed{2}$. Thus Maple 17 must build a `SUM` and a `PROD` for each term in f whereas Maple 16 only builds a `SUM` and the `PROD` already exists. Theoretically, Maple 17 is $O(nt)$ compared with $O(t)$ for Maple 16. We have tried to improve the speed of unpacking by creating the `PROD` objects in simplified form, but the slowdown for `op(f)` remains a factor of 4.

However, we observed a tendency of Maple library code to either frequently unpack POLY or rarely unpack, often in the course of checking high level types. Alongside other internal caches in the Maple kernel (e.g. for `subs` and `indets`) we added a cache for unpacked POLY dags. This cache is cleared out on every garbage collection, so its practical effect is restricted to small polynomials that would be repeatedly unpacked.

For `indets(f,t)` we can avoid unpacking in many cases by detecting types that do not appear in POLY or appear only in the variables. Table 4 shows the most common types in `indets(f,t)` when f is a POLY when the entire Maple library test suite is run. For products or powers we must unpack the terms of f to create the result, but in the top ten cases (and many others) that is avoided.

Table 4. Calls to `indets`(f, t) for f a POLY in the Maple library test suite.

type t	number	type t	number
name	11937973	{rtable, table}	1509366
nonreal	7081486	specfunc(anything, RootOf)	1429737
float	6930777	radical	1101539
function	6678146	indexed	1089504
Or(RootOf, radical)	1863699	‘^‘	1047368
{name, function}	1861368	Or(‘+‘,‘*‘,‘^‘)	828257

3.2 Repacking

A number of the Maple kernel operations require us to repack monomials. For example, when adding $x^2 + y^2 + z^2$ and $x^2 + y^2 - z^2$, the result $2x^2 + 2y^2$ does not have the variable z. When Maple simplifies the result it must detect that z is missing and repack the polynomial into $\mathbb{Z}[x, y]$. Repacking also occurs in the `coeff`, `coeffs`, `eval`, `lcoeff`, and `taylor` commands that remove one or more variables, or `expand` and `divide`, which convert polynomials to a common ring. Critically, these operations do not permute the variables; they insert or remove exponent fields and change the sizes of the remaining fields.

In Maple 17 this is coded as a straightforward $O(n)$ loop, but we digress on the topic because new Intel microprocessors (codenamed Haswell, see [13]) have added two new instructions with exciting possibilities. The PEXT instruction is short for parallel extract, and it flushes masked bits to the bottom of the word.

monomial $x^3y^5z^7$	00001111 00000011 00000101 00000111
mask for $\{x, z\}$	00000000 11111111 00000000 11111111
result from PEXT	00000000 00000000 00000011 00000111

Its inverse is PDEP, short for parallel deposit, which distributes bits to various locations in a word, starting from the lowest bit. Both operations are $O(1)$.

input data	00000000 00000000 00000011 00000111
mask of locations	00000000 11111111 00000000 11111111
result from PDEP	00000000 00000011 00000000 00000111

The parallel design for these operations was originally proposed by Hilewitz and Lee in [12]. With these instructions, we can repack monomials without any branches or loops, e.g. to convert from $\mathbb{Z}[x, z]$ to $\mathbb{Z}[x, y, z]$ we would do:

$$\begin{array}{ccccc} 3 \times 21 \text{ bits} & \textit{extract} & 4 \times 16 \text{ bits} & \textit{deposit} & 4 \times 16 \text{ bits} \\ \begin{array}{|c|c|c|}\hline d & i & k \\ \hline\end{array} & \longrightarrow & \begin{array}{|c|c|c|c|}\hline 0 & d & i & k \\ \hline\end{array} & \longrightarrow & \begin{array}{|c|c|c|c|}\hline d & i & 0 & k \\ \hline\end{array} \end{array}$$

This would be needed, for example, to multiply $f(x, z)$ and $g(x, y)$ in $\mathbb{Z}[x, y, z]$.

In fact Maple 17 does use the PEXT operation to reorder multiple variables for `coeffs`, but we coded this in C using the algorithm in Hacker's Delight [21]. After precomputation it uses 24 bit instructions per word, and it is called twice per term by `coeffs` to reorder the exponents. We attemped to use a C routine for PDEP to repack monomials but it was hard to get a gain. Nevertheless this should be our approach as soon as there is widespread hardware support.

3.3 Hashing and Simplification

When an algebraic expression is created in Maple, it is simplified recursively by the kernel. For example, consider the polynomial from Figure 1.

> f := $9xy^3z - 4y^3z^2 - 6xy^2z - 8x^3 - 5$;

Each object in f that is not a small integer is simplified and hashed to see if it already exists in memory. In Maple 16, this is first done for the variables x, y, z then for the monomials (`PROD`s in Figure 1) xy^3z, y^3z^2, xy^2z and x^3, and finally for the whole expression f. These objects are hashed and stored in the internal `simpl` table, which maps each object to a unique copy in memory. This feature of Maple allows it to identify equal objects by address.

In Maple 17 (see Figure 3), because monomials are encoded immediately as machine integers, they are not stored in the `simpl` table. Rather, only the `POLY` object, the `SEQ` object (the sequence of varibles), and multiprecision coefficients are stored in the `simpl` table.

What is gained from not having to create, simplify, and hash each monomial as a `PROD` object? The following benchmark gives us a clue. Consider the input

> f := expand($(1+s+t+u+v+w+x+y+z)^{16}$):

which creates a polynomial f in 8 variables with 735471 terms. By first issuing the command `sdmp:-info(1):` we can obtain profiling information from the C library that computes this result. Table 5 shows that most of the time is spent simplifying the result in Maple 16, whereas in Maple 17 this time is fairly small. The C routine itself is identical with only a tiny difference due to the compiler.

Table 5. Real time in seconds for computing and simplifying a large power.

	expand power	allocate dag	simplify dag	total time
Maple 16	0.133 s	0.080 s	1.180 s	1.420 s
Maple 17	0.128 s	0.000 s	0.010 s	0.139 s

The C library uses the same monomial representation as Maple 17, so it can copy the term data to a new POLY object. Maple 16 has to allocate the sum of products structure, which is almost as expensive as the computation. Maple 17 simplifies the POLY by checking that its terms and variables are sorted and all variables have a non-zero exponent. This takes 0.01 seconds or 7.2% of the time. Maple 16 must do considerably more work to simplify the sum of products dag. For each `PROD`, it checks that the variables are distinct (they are) using an $O(n^2)$ loop, then it hashes the `PROD` and inserts it into the `simpl` table. Then it has to sort the `SUM` to check that all the `PROD`s are distinct (they are), because hashing has destroyed any previous order. Finally, it hashes the `SUM`.

A hidden cost is that the code to simplify `SUM`s and `PROD`s is quite expensive. It handles special objects like *infinity* and *undefined*, complex and floating point arithmetic, operator overloading, binary relations like $<$ or $=$ because equations can be added or scaled, matrix arithmetic, etc. These routines implement much of the expressive power of the Maple language, and this is not free. To simplify `POLY`, we have the luxury of analyzing the variables and calling algorithms that work in restricted domains. But in atypical cases, we unpack `POLY` and fall back on the existing code for `SUM` and `PROD`.

4 Benchmarks

What is the impact of the POLY data structure on Maple's overall performance? This was difficult to predict in advance. One goal was to reduce sequential overhead in polynomial algorithms so that parallel speedup in multiplication and division (see [18, 19]) would speed up the Maple library. Was that achieved? To this end we developed two benchmarks; expanding determinants of polynomial matrices and factoring multivariate polynomials. Both are higher level algorithms.

4.1 Determinant Benchmark

Our first benchmark computes the determinant of the $n \times n$ symmetric Toeplitz matrix A for $6 \leq n \leq 11$. This is a matrix with n variables $\{x_1, \ldots, x_n\}$ with x_i along the i^{th} diagonal and i^{th} subdiagonal. To compute $\det(A)$ we use our own implementations of the Bariess algorithm [2], which we provide in the appendix. At the k^{th} elimination step, ignoring pivoting, the Bareiss algorithm computes

$$A_{i,j} := \frac{A_{k,k}A_{i,j} - A_{i,k}A_{k,j}}{A_{k-1,k-1}} \quad \text{for} \quad i = k+1, \ldots, n \quad \text{and} \quad j = k+1, \ldots, n \qquad (1)$$

where the division is exact. At the end of the algorithm $A_{n,n} = \pm \det(A)$. Thus the Bariess algorithm does a sequence of $O(n^3)$ polynomial multiplications and divisions that grow in size, with the largest one occurring in the last step when $k = n - 1$.

In Table 6 below, #det is the number of terms in the determinant which has degree n, and #num is the number of terms in $A_{n-1,n-1}A_{n,n} - A_{n,n-1}A_{n-1,n}$ which has degree $2n - 2$ and is much larger than $\det(A)$.

We used a quad core Intel Core i7 920 2.66 GHz CPU running 64-bit Linux. Timings are real times in seconds. With four cores we achieve a factor of 3 to 4 speedup over Maple 16, which is large. That gain is entirely from reducing the overhead of Maple data structures; there is no change in polynomial arithmetic versus Maple 16, which the same C library routines. The reduction of overhead increases parallel speedup from 1.6x to 2.59x over Maple 16. For comparison we include times for Maple 13 (which does not use our C library) and Magma 2.17.

Table 6. Real time in seconds to compute $\det(A)$ using the Bareiss algorithm.

			Maple 13	Maple 16		Maple 17		Magma 2.17
n	#det	#num	1 core	1 core	4 cores	1 core	4 cores	1 core
6	120	575	0.015	0.008	0.009	0.002	0.002	0.000
7	427	3277	0.105	0.030	0.030	0.006	0.006	0.020
8	1628	21016	1.123	0.181	0.169	0.050	0.040	0.200
9	6090	128530	19.176	1.450	1.290	0.505	0.329	2.870
10	23797	813638	445.611	14.830	12.240	6.000	3.420	77.020
11	90296	5060172	–	151.200	94.340	88.430	34.140	2098.790

By default, Maple and Magma do not use the Bareiss algorithm to compute these determinants. Instead, they use the method of minor expansion presented by Gentleman and Johnson in [7]. Recall that given an $n \times n$ matrix A

$$\det(A) = \sum_{i=1}^{n} (-1)^{n+1} A_{i,1} \det(M(1,i)) \tag{2}$$

where $M(1,i)$ is the $n-1$ by $n-1$ matrix obtained from A by deleting column 1 and row i. Applied naively, this formula recomputes the determinants of sub-matrices many times. Gentleman and Johnson avoided that by computing from the bottom up; they compute all 2×2 determinants then all 3×3 determinants and so on. This is still exponential in n. It computes $\binom{n}{k}$ determinants of $k \times k$ sub-matrices for a total of $\sum_{k=1}^{n} \binom{n}{k} = 2^n - 1$ determinants.

For our Toeplitz matrices, the multiplications in (2) are of the form *variable times polynomial.* For example, to multiply $f = 9xy^3z - 4y^3z^2 - 6xy^2z - 8x^3 - 5$ by y, we add the monomial representation for y to each monomial in f, namely $y = \boxed{\texttt{1010}} = 2^{48} + 2^{16}$ to the encodings of $\boxed{\texttt{5131}}$, $\boxed{\texttt{5032}}$, $\boxed{\texttt{4121}}$, $\boxed{\texttt{3300}}$, $\boxed{\texttt{0000}}$. Notice how the result remains sorted in the monomial ordering. The additions in (2) are n-ary, for which we copy terms to a new POLY and apply radix sort. The improvement shown in Table 7 was huge. It surprised us.

Table 7. Real times in seconds for determinants using minor expansion.

n	#det	Maple 16	Maple 17	Magma 2.17
6	120	0.002	0.002	0.001
7	427	0.010	0.004	0.003
8	1628	0.049	0.013	0.019
9	6090	0.305	0.047	0.116
10	23797	1.991	0.252	0.770
11	90296	19.370	1.322	6.210
12	350726	274.990	6.737	44.500
13	1338076	2024.370	37.570	337.770

4.2 Factorization benchmark

Our second benchmark is multivariate factorization. For perspective we include timings for Magma [3], Mathematica, Maxima [14], Sage [20], Singular [10], and Trip [5] which is a computer algebra system for celestial mechanics.

Table 8 reports the real times for multiplication, division, and factorization on a hyperthreaded quad core Intel Core i7 920 2.66 GHz running 64-bit Linux. For each timing we report the median of three test runs. Maple 16 and 17 start up to four threads depending on the size of each multiplication or division. The factorization routine is sequential Maple code, which gains parallelism from our multiplication and division routines.

For Mathematica 9 we timed the internal functions `Algebra‘IPExpand` and `Algebra‘IPExactQuotient` for multiplication and division. The additional time for the top level `Expand` was small, and we found no suitable command for exact division at the top level. The `Factor` command did not make use of parallelism.

We report two timings for Trip: the (RS) time is for the optimized recursive sparse polynomial data structure POLYV, while the (RD) time is the optimized recursive dense data structure POLPV. Both use Trip's parallel routines (see [6]) with 8 threads and rational arithmetic, including a fast representation for small machine integers similar to Maple's.

There are some anomalies in Table 8. Maple's times for division are close to those for multiplication, except on Problem 5 where Maple 17 uses a sequential dense method to multiply. Singular's timings for p_3/f_3 and p_4/f_4 are over twice as fast as the times for multiplication. This is because Singular multiplies in the distributed representation and divides in a recursive representation. In contrast, Trip's times for division are slower than those for multiplication, partly because division in Trip 1.2 is not parallelized.

In comparing the timings for factoring p_1 and p_2 we see that factoring p_2 is much slower in Maple 13 and Mathematica, but not in Maple 16 or 17, Magma, or Singular. The fast systems apply the substitution $p_2(x^2 = u, y^2 = v, z^2 = w)$ to reduce the degree of the input polynomial before factoring it. This halves the number of Hensel lifting steps in each variable.

We note that Singular's timings for factorization have improved enormously since version 3-1-0, Times for version 3-1-0 on the first four factorizations were 12.28, 23.67, 97.10, 404.86 seconds. The factorization code was changed to use a recursive representation for polynomials by Michael Lee.

Our first improvement from Maple 13 to Maple 16 was due to our C library for polynomial multiplication and division described in [16–19] and reported at ISSAC 2010. The speedup in multiplication and division produces a speedup in multivariate factorization, because most of the time is spent in "Hensel lifting" which consists of many multiplications and some exact divisions.

Our second improvement was to parallelize the algorithms for multiplication and division. In many cases we obtain superlinear speedup in our C library but the top level `expand` and `divide` have lower speedup, because of the extra time to import and export Maple data structures. For higher level algorithms such as `factor`, parallel speedup is further reduced by the need to perform many small

Table 8. Real times in seconds for polynomial multiplication, division and factorization on an Intel Core i7 920 2.66GHz quad core desktop running Linux.

	Maple 13	Maple 16		Maple 17		Magma	Sage	Singular	Maxima	Mathematica		Trip 1.2.26	
		1 core	4 cores	1 core	4 cores	2.19-1	5.8	3-1-6	5.25.0	7.0	9.0	(RS)	(RD)
multiply													
$p_1 := (f_1+1)(f_1+2)$	1.561	0.063	0.030	0.041	0.012	0.330	1.090	0.585	0.56	4.79	0.120	0.010	0.008
$p_2 := (f_2+1)(f_2+2)$	1.544	0.063	0.032	0.041	0.012	0.330	1.080	0.608	0.57	5.06	0.115	0.024	0.016
$p_3 := (f_3+1)(f_3+2)$	26.501	0.521	0.171	0.403	0.102	3.990	10.860	6.595	24.25	50.36	0.855	0.093	0.083
$p_4 := (f_4+1)(f_4+2)$	98.351	2.180	0.649	1.814	0.416	13.700	61.770	31.806	5.83	273.01	5.732	0.501	0.396
$p_5 := f_5 \cdot g_5$	13.666	1.588	0.384	0.153	0.154	13.240	32.730	17.776	15.83		1.526		
$p_6 := f_6 \cdot g_6$	11.486	0.772	0.628	0.204	0.082	0.890	3.040	1.787	2.52				
divide													
$q_1 := p_1/f_1$	1.451	0.065	0.033	0.042	0.015	0.360	1.300	0.183	0.55	6.09	0.197	0.296	0.208
$q_2 := p_2/f_2$	1.435	0.065	0.033	0.042	0.015	0.360	1.300	0.183	0.56	6.53	0.194	0.293	0.225
$q_3 := p_3/f_3$	25.054	0.524	0.184	0.411	0.117	4.140	17.810	4.737	24.63	46.39	1.510	2.490	1.880
$q_4 := p_4/f_4$	92.867	2.253	0.736	1.842	0.483	18.540	80.390	11.420	5.93	242.87	5.662	9.880	6.100
$q_5 := p_5/f_5$	5.570	1.636	0.417	1.445	0.333	12.480	50.160	10.478	16.14		2.789		
$q_6 := p_6/f_6$	10.421	0.769	0.627	0.215	0.095	7.900	4.870	1.484	2.69				
factor													
p_1 : 12341 terms	31.330	2.792	2.658	0.790	0.650	6.510	1.510	0.853	4.54	11.82	18.478		
p_2 : 12341 terms	275.508	3.240	3.071	0.991	0.850	7.090	1.580	0.933	4.97	64.31	112.243		
p_3 : 38711 terms	360.862	16.714	14.110	6.927	4.399	119.320	18.140	10.074	163.06	164.50	276.161		
p_4 : 135751 terms	2856.388	59.009	46.151	24.345	12.733	320.040	68.320	39.353	44.94	655.49	951.725		
p_5 : 12552 terms	302.453	26.435	16.152	12.131	6.800	105.550	14.630	9.604	1046.67		935.149		
p_6 : 417311 terms	1359.473	51.702	48.808	8.295	6.330	369.120	37.560	20.603	155.49				

$f_1 = (1+x+y+z)^{20}$
1771 terms

$f_2 = (1+x^2+y^2+z^2)^{20}$
1771 terms

$f_3 = (1+x+y+z)^{30}$
5456 terms

$f_4 = (1+x+y+z+t)^{20}$
10626 terms

$$f_5 = (1+x)^{20}(1+y)^{20}(1+z)^{20}+1$$
$$g_5 = (1-x)^{20}(1-y)^{20}(1-z)^{20}+1$$
9261 terms

$$f_6 = (1+u^2+v+w^2+x-y)^{10}+1$$
$$g_6 = (1+u+v^2+w+x^2-y)^{10}+1$$
3003 terms

operations in sequence. The cost of `degree`, `indets`, and `type(f,polynom)` also reduce parallelism in higher level code.

With the introduction of the POLY dag in Maple 17, we have substantially reduced data structure overhead and the cost of almost all supporting routines. Table 9 shows the improvements to parallel speedup which come on top of the large gains achieved for sequential time. The speedup for `expand` and `divide` is now much closer to our C library, and the speedup for `factor`, while modest, is respectable for a sequential algorithm.

Table 9. Parallel speedup (1 core)/(4 cores) in Maple 17 versus Maple 16.

	Maple 17						Maple 16					
expand	3.41	3.41	3.95	4.36	1.00	2.49	2.10	1.97	3.04	3.36	4.13	1.23
divide	2.80	2.80	3.51	3.81	4.34	2.26	1.97	1.97	2.85	3.06	3.92	1.23
factor	1.21	1.16	1.57	1.91	1.78	1.31	1.05	1.05	1.18	1.28	1.63	1.06

Table 10. Observed speedup (Maple 16)/(Maple 17) on 4 cores versus 1 core.

	4 cores						1 core					
expand	2.50	2.66	1.67	1.56	2.49	7.65	1.53	1.53	1.29	1.20	10.38	3.78
divide	2.20	2.20	1.57	1.52	1.25	6.60	1.54	1.54	1.27	1.22	1.13	3.57
factor	4.09	3.61	3.20	3.62	2.37	7.71	3.53	3.27	2.41	2.42	2.18	6.32

Table 10 shows the speedup of Maple 17 over Maple 16, on 4 cores versus 1. Notice how the gain is larger in parallel. This is just a pleasant consequence of Amdahl's Law when you reduce sequential overhead in parallel algorithms. The sole exception (Problem 5 expand) uses a sequential dense method in Maple 17.

With the POLY dag in Maple 17, the time for factorization on one core has been reduced by more than 50%, but the parallel speedups are even greater:

The savings are entirely sequential time, as can be seen by subtracting the parallel times from the sequential times. The reduction in overhead improves parallel speedup significantly, from 1.3x to 2.0x in the case of factoring p_4.

We note that Maple's factorization code has not changed since 1984. However, there is little parallel speedup. We achieve significant additional speedup (compare Maple 16 with the new POLY dag) with the POLY dag used by default. For factoring p_4 we obtained a sequential improvement of a factor of 53.52/26.43 = 2.02× and a parallel improvement of a factor of 44.84/16.17 = 2.77×. Parallel speedup for factoring p_4 improved from 53.52/44.84 = 1.19× to 26.43/16.17 = 1.63× in our new Maple.

A closer examination of the timings shows that parallel speedup for the multiplication $p_4 \times (p_4 + 1)$, which is a factor of 1.810/0.632 = 2.76× is still quite poor even though our parallel C library for multiplication is 4 times faster on the actual multiplication. Why is this? There are two reasons. One is that on the Core i5, if one uses one core only, that core will run in *turbo boost* mode which on our Core i5 is $\leq$ 3.20Ghz/2.66Ghz = 1.2$\times$ faster. The other reason is the sequential overhead in the integration of our parallel multiplication and division software. For a polynomial multiplication $c := a \times b$, Maple 16 first converts the input polynomials a and b from the sum-of-products data structure to our POLY data structure, then multiplies them using our external parallel C library, which does achieve a factor of 4 speedup on 4 cores, then converts the product c back to Maple's sum-of-products data structure. There is additional sequential overhead required to determine how many words are required to pack the monomials. Maple must compute the union of the sets of variables in a and b and the total degree of a and b in those variables. This involves many passes through the sum-of-products data structures for a, b and c. This overhead is largely eliminated in the new Maple. Adjusting for the turbo boost the parallel speedup in the new Maple for 4 cores is $\frac{1.73}{0.508} \times \frac{3.20}{2.66} = 4.09\times$.

To see where the improvements in the factorization have come from we have profiled the main parts of the factorization code. The profile (see Table 6) shows the %age of the time in the main parts of the factorization algorithm for Maple 16 and our new Maple. The data under *improved coeftayl* includes a further algorithmic improvement. The data shows we have eliminated $0.599 - 0.377 = 0.222s$ of overhead from the polynomial multiplications (see row `expand`) or 37%. The biggest speedup is division (see row `divide`). This is because the divisions are mostly trial divisions which fail quickly. In such cases almost all the time is in conversion which is wasted.

	Maple 16		New Maple		improved coeftayl	
function	time	time%	time	time%	time	time%
coeftayl	1.086s	41.06	0.310s	28.21	0.095s	12.03
expand	0.506s	19.13	0.263s	23.93	0.255s	32.28
diophant	0.424s	16.03	0.403s	34.94	0.299s	37.85
divide	0.256s	9.68	0.034s	3.09	0.035s	4.43
factor	0.201s	7.60	0.011s	1.00	0.010s	1.27
factor/hensel	0.127s	4.80	0.064s	5.82	0.063s	7.97
factor/unifactor	0.045s	1.70	0.033s	3.00	0.033s	4.18
total:	2.645s	100.00%	1.099s	100.00%	0.790s	100.00%

Table 11. profile for `factor(p1);` (1 core).

The biggest absolute gain is for the routine `coeftayl(f,x-a,k)` which computes the coefficient of f in $(x-a)^k$. This computation is not done by expanding f as a Taylor series about $x = a$ but rather by using the formula $g(x = a)/k!$ where $g = \frac{\mathrm{d}f}{\mathrm{d}^k x}$, the k'th derivative of f. Referring back to Table 1, we can see that the speedup is due to the improvement of differentiation and polynomial evaluation. We also tried the following formula to compute the coefficient: $\sum_{i=k}^{\deg_x f} \operatorname{coeff}(f, x^i) a^i \binom{i}{k}$. We can see that this is 3$\times$ faster again (see improved coeftayl). The total real time is reduced from 2.59s to 1.07s to 0.790s.

5 Conclusion

Maple, Mathematica, Magma and Singular all use a distributed representation for multivariate polynomials. Maple's sum-of-products data structure and Singular's linked list data structure are illustrated in Figures 1 and 2 in the introduction. We ask the the reader take another good look at them. Mathematica's data structure is similar to Maple's and Magma's data structure is similar to Singular's. These data structures, which were designed in the 1980s when memory access was constant time, will not yield high-performance on todays computers because memory access is not sequential.

One way to speed up polynomial multiplication, division, or factorization would be to convert the input to a more suitable data structure, compute the result, then convert back. This is what we did in [16] for Maple 14 for polynomial multiplication and division. Singular 3-1-4 does this for polynomial division and factorization. It switches to using a recursive representation for division and factorization. However, the conversion overhead will limit parallel speedup. Ahmdah's law states that if the sequential proportion of a task is S then parallel speedup on N cores is limited to

$$\textit{speedup} \leq \frac{1}{S + (1-S)/N}.$$

When S is large (50% or more say), then we cannot get good parallel speedup.

What we have done in this work for Maple is to make our POLY data structure the default data structure in Maple. The POLY data structure is used when all monomials in a polynomial can be packed into a single word. This enabled us to eliminate conversion overhead in multiplication and division. The data in Table 5 shows improved parallel speedup for polynomial multiplication and division. We also implemented highly efficient algorithms for many Maple kernel operations for POLY. The data in Table 4 shows a speedup of a factor of 50 over Maple 16 for a routine polynomial determinant computation. The data in Table 5 shows speedups of factors of between 2 and 3 for large multivariate polynomial factorizations which is a huge gain. Although not reported here, we also find speedups of a factor of 2 for large multivariate polynomial gcd computations.

The cost incurred is mainly in code complexity. We must manage two data structures for polynomials, one where the coefficients are integers and the monomials can be packed into a single machine word, and one, Maple's sum-of-products data structure, which does not have these restrictions. A substantial programming effort was required to support the new data structure in the Maple kernel. The gains suggest this is worthwhile.

In closing, the reader may have wondered why we only use one word of memory to encode monomials, and not two, or more? Afterall, if we use two words, we could encode polynomials in twice as many variables or of much higher degree. With 128 bits, one will cover almost all applications. We would like to see how far 64-bits takes us before considering such an extension. For supporting two word exponents potentially doubles the amount of code. Another desirable extension is to allow the coefficients in the POLY dag to be fractions or floating point numbers as well as integers.

References

1. O. Bachmann and H. Schönemann. Monomial representations for Grobner bases computations. *Proceedings of ISSAC '98*, pp. 309–316, 1998.
2. E. Bariess, 1968. Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination. *Mathematics of computation* **22** (102): 565−578.
3. Bosma, W., Cannon, J., Playoust, C., 1997. The Magma Algebra System I: The User Language. *J. Symb. Cmpt.* **24**(3-4), 235–265. See also `http://magma.maths.usyd.edu.au/magma`
4. Buchberger, B. Private Communication, May 2013.
5. Gastineau, M., Laskar, J., 2006. Development of TRIP: Fast Sparse Multivariate Polynomial Multiplication Using Burst Tries. *Proceedings of ICCS 2006*, Springer LNCS **3992**, pp. 446–453.
6. Gastineau, M., 2010. Parallel operations of sparse polynomials on multicores - I. Multiplication and Poisson bracket. *Proceedings of PASCO '2010*, ACM Press, pp. 44–52, 2010.
7. Gentleman, W.M., Johnson, S.C. Analysis of Algorithms, A Case Study: Determinants of Matrices with Polynomial Entries. *ACM Trans. on Math. Soft.*, **2**(3), pp. 232–241, September 1976.
8. Granlund, T., 2008. The GNU Multiple Precision Arithmetic Library, version 4.2.2. `http://www.gmplib.org/`
9. Grayson, Daniel R. Stillman, Michael E., Macaulay2, a software system for research in algebraic geometry. Available at http://www.math.uiuc.edu/Macaulay2/
10. Greuel, G.-M., Pfister, G., Schönemann, H., 2005. Singular 3.0: A Computer Algebra System for Polynomial Computations. Centre for Computer Algebra, University of Kaiserslautern. http://www.singular.uni-kl.de
11. Hall, A.D. Jr., The ALTRAN System for Rational Function Manipulation – A Survey. *Communications of the ACM*, **14**, 517–521, ACM Press, 1971.
12. Hilewitz, Y., Lee, R.B., 2006. Fast Bit Compression and Expansion with Parallel Extract and Parallel Deposit Instructions. *Proceedings of ASAP '06*, IEEE, pp. 65–72, 2006.
13. Intel Corporation. Advanced Vector Extensions Programming Reference. June 2011. `http://software.intel.com/sites/default/files/m/8/a/1/8/4/36945-319433-011.pdf`
14. The Maxima computer algebra system. http://maxima.sourceforge.net/
15. Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. Engineering Radix Sort, Computing Systems, **6**(1): 5–27, 1993.
16. Monagan, M., Pearce, R., 2010. Sparse Polynomial Multiplication and Division in Maple 14. *Communications in Computer Algebra*, **44**:4, 205–209, December 2010.
17. Monagan, M., Pearce, R., 2011. Sparse Polynomial Division using Heaps. *J. Symb. Cmpt.* **46**(7):807–822, 2011.
18. M. Monagan, R. Pearce., 2009. Parallel Sparse Polynomial Multiplication Using Heaps. *Proceedings of of ISSAC 2009*, ACM Press, pp. 295–315.
19. M. Monagan, R. Pearce., 2010. Parallel Sparse Polynomial Division Using Heaps. *Proc. of PASCO 2010*, ACM Press, pp. 105–111.
20. William A. Stein et al. Sage Mathematics Software (Version 5.0), The Sage Development Team, 2012, http://www.sagemath.org.
21. Warren, Henry S. Hacker's Delight. Addison-Wesley, 2003.

Appendix A

Maple code (no pivoting) for the Bareiss algorithm.

```
ffge := proc(A,n) local d,i,j,k,t;
   d := 1;
   for k to n-1 do
       for i from k+1 to n do
           for j from k+1 to n do
               t := expand(A[k,k]*A[i,j]-A[i,k]*A[k,j]);
               divide(t, d, evaln(A[i,j]));
           od;
           A[i,k] := 0;
       od;
       d := A[k,k];
   od:
   A[n,n];
end;
n := 8;
T := linalg[toeplitz]([seq(x[i],i=1..n)]);
A := array(1..n,1..n):
for i to n do for j to n do A[i,j] := T[i,j] od od:
det := CodeTools[Usage]( ffge(A,n) ):
```

Magma code for the Bareiss algorithm.

```
Z := IntegerRing();
P<x,y,z,u,v,w,p,q,r,s,t,a> := PolynomialRing(Z,12);
X := [x,y,z,u,v,w,p,q,r,s,t,a];
n := 8;
A := Matrix(P,n,n,[0 : i in [1..n^2]]);
for i in [1..n] do
    for j in [1..n] do
        A[i,j] := X[AbsoluteValue(j-i)+1];
    end for;
end for;
d := 1;
time for k in [1..n-1] do
         for i in [k+1..n] do
            for j in [k+1..n] do
                t := A[k,k]*A[i,j]-A[i,k]*A[k,j];
                A[i,j] := ExactQuotient(t,d);
            end for;
         end for;
         d := A[k,k];
     end for;
det := A[n,n];
```

Appendix B

Maple code for timing benchmarks.

```
f := expand( (1+x+y+z)^20 )+1:
p := CodeTools[Usage]( expand( f*(f+1) ) ):
CodeTools[Usage]( divide(p,f,'q') );
CodeTools[Usage]( factor(p) ):
```

Magma code for timing benchmarks.

```
Z := IntegerRing();
P<x,y,z> := PolynomialRing(Z,3);
f := (1+x+y+z)^20+1;
g := f+1;
time h := f*g;
time q := ExactQuotient(h,f);
time ff := Factorization(h);
```

Mathematica code for timing benchmarks.

```
f = Expand[(1+x+y+z)^20]+1;
AbsoluteTiming[p = Expand[f*(f+1)];]
AbsoluteTiming[q = PolynomialQuotient[p,f,x];]
AbsoluteTiming[h = Factor[p];]
```

Maxima code for timing benchmarks.

```
showtime : true;
f : rat( (1+x+y+z)^20 ) +1 $
h : f*(f+1)$
qr : divide( h, f )$
f : factor( h )$
```

Sage code for timing benchmarks

```
Q = RationalField()
P.<x,y,z> = PolynomialRing(Q,3,order='deglex')
f = (1+x+y+z)^20+1
%time p = f*(f+1)
%time q,r = p.quo_rem(f)
%time h = factor(p)
```

Singular code for timing benchmarks.

```
ring R=0,(x,y,z),lp;
poly f = (1+x+y+z)^20+1;
poly g = f+1;
int TIMER;
TIMER = timer; poly p = f*g; timer-TIMER;
TIMER = timer; poly q = p/f; timer-TIMER;
TIMER = timer; list L = factorize(p); timer-TIMER;
```

Trip code for timing benchmarks. POLYV means recursive sparse, POLPV means recursive dense.

```
reset; _cpu=4$ _mode=POLYV$ _modenum=NUMDBL$
f=(1+x+y+z)^20+1$ g=f+1$ p = 0$
time_s; p = f*g$ time_t(usertime, realtime); realtime;
time_s$ div(p,f,q,r)$ time_t(ctime,rtime)$ rtime;
```