DRAFT: Comparing the speed of programs for sparse polynomial multiplication

Richard Fateman

July 24, 2002

Abstract

How should one design and implement a program for the multiplication of sparse polynomials? This is a simple question, necessarily addressed by the builders of any computer algebra system (CAS). To examine a few options we start with a single easily-stated computation which we believe represents a useful benchmark of "medium difficulty" for CAS designs.

Introduction: a specific task

Consider the expanded version of the polynomial $p^{20} = (1+x+y+z)^{20}$. This polynomial has 1771 distinct monomial terms. By one ordering, the first term is z^{20} and the 885th term is $1163962800x^9y^5z^4$. Its largest coefficient is 11732745024 which requires more that one 32-bit word to store, since it exceeds 2^{31}. The size of the polynomial p^n grows like this:

n	terms
1	4
2	10
4	35
8	165
16	969
20	1771
40	12341

That is, squaring p^{20} explicitly yields 12,341 terms.

A clever CAS will try to avoid computing such an expansion since the factored form is so much more compact. Nevertheless, sometimes explicit multiplications are necessary as part of a sequence of operations including adds and cancellations. We have referred to this polynomial sequence as "sparse" since in each case the number of terms in p^n is substantially smaller than the maximum number of terms in a polynomial of the same maximum degrees. It is generally believed that polynomials in a CAS are likely to be rather sparse.

The rest of this paper describes how we can compute this product in various ways.

Some representations

Several sparse representations can be conveniently expressed in Common Lisp.

The computer algebra system Macsyma represents this kind of polynomial as a list of lists.

We have somewhat simplified the expression format but we use for an example the expansion of $(x+y+z+1)^2$:

```
(z 2 1 1 (y 1 2 0 (x 1 2 0 2)) 0 (y 2 1 1 (x 1 2 0 2) 0 (x 2 1 1 2 0 1)))
```

which decodes to the *sparse recursive form*

$$z^2 * 1 + z^1 * (y^1 * 2 + y^0 * (x^1 * 2 + x^0 * 2)) + z^0 * (y^2 * 1 + y^1 * \cdots))$$

The computer algebra system MockMMA [2] which we wrote a number of years ago uses a similar representation in that it is recursive,

```
 #(z #(y #(x 1 2 1) #(x 2 2) 1) #(y #(x 2 2) 2) 1)
```

except that it uses *vectors* and (an inesssential detail) the terms are given in reverse order. The Lisp notation `#(a b c)` is the print form for a vector of three items. Contrary to the popular notion of lisp making everything out of linked lists, this vector is represented as three adjacent cells in memory: there are no pointers. In addition to not "wasting" space on links, MockMMA's vector representation omits the exponents. This means that non-leading zero coefficients take space. This is a *dense recursive form* where the density is reconsidered at each recursive level.

To see how this trade-off works, for the expression $43x^9+5x+1$ Macsyma would use a list (x 9 43 1 5 0 1) and MockMMA would use the vector `#(x 1 5 0 0 0 0 0 0 0 43)`. For the expression $43x^2+25x+10$ Macsyma would use a list (x 2 43 1 25 0 10) and MockMMA would use the vector

`#(x 10 25 43)`. Sometimes one form is more compact and sometimes the other.

Either the list or vector representation has to allow for the possibility of coefficients like 43 or 9 in the examples above to be *arbitrary-precision integers* or *bignums.* This potential complication results in considerable expense compared to programs written with the *a priori* restriction that all coefficients are short integers, integers modulo a small prime, or some fixed-size floats.

A third representation corresponds to a collection of monomials in a hash table. We can convert to this representation by traversing one of the other representations and successively encoding each of the monomials. We collect exponents and pack them into a key. The value is the associated coefficient. In the same example shown above, $\{1 * z^2y^0x^0,\ 2 * zy^2x^0,\ 2 * z^1y^1x^1,\ ...\}$ would be put in the hash table. Here we propose to "compress" the exponents by computing a single key: for the term $c * z^ny^mz^k$ the key could be $n + 100m + 10000k$, and the coefficient c would be stored in the table at that key location. This works as long as the polynomial degrees needed are not too high (in this case more than 100) and it is especially convenient if this combined key is a small integer (not a bignum). The exponent spacing of 100 can be altered to accomodate higher degrees, but the bound must be anticipated in some way. This idea of packing exponents subject to some pre-declared upper bound was used in ALTRAN [1], an early computer algebra system. It is not quite as general as we would like, but is not too burdensome to anticipate an upper bound on degrees; a careful implementation (as in ALTRAN) can even detect overflows by padding the bit fields.

A final alternative worth considering is a vector: adjacent memory locations (like the MockMMA representation) but with explicit exponent and coefficient pairs. That is, no overhead for pointers. It will turn out that this representation *unordered* has the advantage of compactness over the hashtable, and for multiplication, this may not be bad at all. On the other hand, we may prefer to eventually sort this vector by exponent, since the cost may be modest and we would certainly benefit by this sorting if we were adding polynomials.

1 Algorithms

Fundamentally, the best we know how do do sparse algorithm multiplication does $O(st)$ coefficient multiplications and $O(st-k)$ additions where the input polynomials have s and t terms and the result has k terms.

Consider an attempt to improve on this by using dense methods: an FFT method, or Karatsuba multiplication. These will not provide faster results because these methods will generally fill in the terms. Our input polynomial $p^{20} = (1 + x + y + z)^{20}$, if represented as "dense to degree 20" in 3 variables would consist of 10,460,353,203 terms instead of the actual 1,771 terms.

A more detailed example is probably justified here. Karatsuba multiplication works best if you have two dense polynomials of the same degree in some variable, say z, of degree $2n$. Consider splitting p as $A * z^n + B$ where A and B are of now of degree n or less in z and similarly split q into $C * z^n + D$. With a naive algorithm we would multiply $p * q$ at cost $size(p) * size(q)$, to get $ACz^{2n} + (AD + BC)z^n + BD$. But the Karatsuba algorithm first computes $t_1 = (A + B) * (C + D)$, $t_2 = A * C$, $t_3 = B * D$, and then $t_4 = t_1 - t_2 - t_3 = AD + BC$, to get $t_2 * z^{2n} + t_4 * z^n + t_3$. The Karatsuba algorithm is used recursively for the three multiplications, and for a dense polynomial would use $O(n^{1.5})$ rather than $O(n^2)$ multiplications. The algorithm's speed depends on the fact that $A + B$, being half the degree of p, is about half the size. Unfortunately if p is sparse, *$A + B$ is likely to have about as many terms as p.* That means we have reduced the original problem to three multiplications problems, *but one of them is about as difficult as the original one.* In our benchmark problem the size of $A + B$ is 1486, much larger than 1771/2. For our benchmark, Karatsuba multiplication takes more than 4,495,217 coefficient multiplies compared to the straightforward method which takes about $(1771)^2 = 3,136,441$ multiplies. A similar analysis using the (dense) FFT for multiplying (sparse) polynomials shows that it too is not a good idea.

Given that better algorithm complexity will apparently not provide a lever for faster results, the issue then becomes how to manage storage (especially in the face of cache memory) to optimize this loop: `for each term u in p do for each term v in q do addintosum(u*v)`.

We revisit our task in terms of storage possibilities. We can investigate storing the sum as a list as in the form for Macsyma, using MockMMA's version of a vector, using some union/find structure, a tree, a hashtable, or a more elaborate structure such as cascading hashtables. We evaluate the most promising of these not generally in use, namely hashing.

Another optimization possibility we touch upon here, but explore in greater depth elsewhere, is *blocking*: the double-loop can be broken into blocks so that only part of each polynomial need be in fast memory at any given time.

Considerations of Coefficient Arithmetic

We are, in a sense, counting coefficient operations. These are not likely to be single arithmetic instructions in our host hardware. If we provide for polynomial operations over integers, rational numbers, finite fields or arbitrary rings (including recursively, polynomials in other variables), where do we check on the coefficient domain? We can try to make the code run faster by duplicating the framework of the routines and building separate particular versions of each program for each domain. These might include “arbitrary size integer coefficients”, “finite field with small modulus (less than a word size)” “polynomials in exactly three variables with integer coefficients ignoring coefficient overflow”, and perhaps others for “finite field with large modulus (more than a word size)”, “single-precision floats”, “double-precision floats”, “integers of no more than 256 bits” etc. Alternatively, we can do what is done in the Macsyma implementation (one of the programs compared below). Macsyma does not have separate polynomial routines over different domains. Instead it repeatedly checks (yes, in the innermost loop doing coefficient arithmetic) to see if a global flag, in this case called `modulus` is set to a small prime, in which case modular arithmetic is performed. The default otherwise is to allow inputs that are arbitrary Common Lisp numbers, which themselves could be integer, rational, single or double floats, small numbers, complex (etc.) So there is an overhead in this system of a type dispatch at every single coefficient operation[1].

The MockMMA and Hashing routines, which are among the systems compared below, are very slightly simpler; they were programmed without consideration for finite-field arithmetic, eliminating the time taken for a “modulus check” but they do still allow for single/ double/ rational/ complex/ bignum arithmetic.

We thought we had a simple experimental setup and were prepared to

[1]When Macsyma’s polynomial code was written (1967) memory was expensive and computers were slow. It was logical to write Macsyma with a certain mixture of concern for speed as well as code-size. Today there are several conflicting trends and design decision might be made in different ways.

Now it is plausible to write a separate module for finite-field arithmetic (maybe for several: GF(2), small (odd) primes, and big primes): memory is now so cheap. It is even plausible to do so automatically by reduplicating code by macro-expansion. On the other hand, in today’s object-oriented world it would be clean and logical to use inheritance, overloaded operators, or generic functions for coefficient operations and use what looks like one piece of code.

Given the total Macsyma context, the time to check the modulus prior to each coefficient operation represents a small overhead, not worth much concern.

compare just three programs, using the same Common Lisp. However, in the course of looking at this task we came across a variety of other computer algebra systems, many of them touting the use of an externally developed multiple-precision arithmetic package as a particular advantage. Since we had no reason to believe that our host Common Lisp was tuned for big-number arithmetic, we tried replacing the coefficient arithmetic with one of the fastest stand-alone arithmetic packages, GMP (Gnu Multiple Precision) version 4.1 for a Pentium-3 architecture. While we are not ready to abandon Lisp's convenient built-in arithmetic and its overall consistent memory management, it is clear that for in-place repetitive arithmetic, a basic process that is needed for polynomial multiplication, GMP provides a facility that for our use in this benchmark can be faster even for 2-word-long arithmetic. For longer integers, GMP's advantage grows. In fact, the operation of "replace x_{i+j} with $x_{i+j} + a_i * b_j$" is not even supported by Lisp unless the numerical result fits into a single word: otherwise the functional model requires allocating a new number object for the result and pointing to it. This is generally slower.

Details of the benchmark, including some other programs

The benchmark task is to take the multivariate polynomial $q = p^{20}$ and multiply it by itself explicitly. It turns out that when we run this benchmark on some other CAS, the program figures out that it is squaring q and thus does less work. In those cases we may actually compute $q * (q + 1)$ and thus forced the expansion.

We timed this computation in MockMMA (recursive vectors), Macsyma (recursive lists), Hashing (using a hash-table for accumulating the intermediate partial sums of the product, using the (built-in) lisp hash-table structure). The Lisp hash-table enlarges automatically if it gets too crowded.

We also tried two additional alternative bignum arithmetic systems. GMP is the Gnu Multiple-Precision package; we tried version 3.1 and 4.1.

Here are the times. (Storage allocation, or garbage collection time (GC) is indicated.)

```
Hashing/GMP4.1,ACL6.1  4.7 sec (no lisp GC)
MockMMA (in ACL 6.1)   5.8 sec (including 2.1 in GC)
Macsyma (in ACL 6.1)   6.9 sec (including 2.0 in GC)
Hashing/GMP3.1,ACL6.1  7.2 sec (no lisp GC)
Hashing (in ACL 6.1)   7.9 sec (including 2.4 in GC)
```

These times were measured on the same machine, a 933 Mhz Pentium 3 with 512 megabytes of RAM running Windows 2000; ACL 6.1 is Allegro Common Lisp version 6.1.

The times above are not perfectly reproducible since each depends to some extent on initial memory configurations. Arguably the times for systems written in Lisp could be compared with the GC times subtracted. It is not that the GC time does not count, just that non-GC CPU time tends to be more reproducible and GC time can generally be reduced substantially (or eliminated) by pre-allocating more memory at the start.

After seeing these results we thought it would be interesting to compare these times to other systems. In particular it would seem to be pointless to continue to compare various systems if there were much faster ones available (and especially if they were similarly general in scope and available free)!

We tried several other Macsyma implementations (different underlying Lisps), and found that a few spent an inordinate amount of time in GC. We also tried some other systems (free and commercial), and are grateful[2] to readers of the newsgroup `sci.math.symbolic` for bringing some of these to our attention and in some cases providing copies to test on our machine. After testing these packages we saw that substituting a non-functional state-changing style of arithmetic using the Gnu Multiple-Precision package might make our program even in Lisp, run faster[3]. We have already shown these results above.

After trying out some additional programs on our machine, we have a larger table:

[2]Well, actually, it kind of blunts the point of this paper as we had initially conceived it, which was to compare lists, vectors, and hashtables, with everything else held constant. It also bloats the paper with footnotes, comments, excuses and program fragments. But it also means we can make some other points beyond our initial goal.

[3]Also a distraction blunting our main point.

```
GP/Pari calc 2.0.17    2.3 sec
MockMMA ACL 6.1/GMP4.1 3.3 sec (including 0.4 in GC)
Fermat (see notes)     4.5 sec
Hashing/GMP4.1,ACL6.1  4.7 sec (no Lisp GC)
Reduce 3.7 (in CSL)    5.0 sec (including 1.3 in GC)
MockMMA (in ACL 6.1)   5.8 sec (including 2.1 in GC)
Singular 2.0.3         6.1 sec
Macsyma (in ACL 6.1)   6.9 sec (including 2.0 in GC)
Hashing/GMP3.1,ACL6.1  7.1 sec (including .04 in GC)
Hashing (in ACL 6.1)   7.9 sec (including 2.4 in GC)
Fermat (std)           8.6 sec (fastest of 2 runs.)
Form 3.0 sorted       21.6 sec (see notes)
MuPad 1.4.2 Pro       29.4 sec (poly form)
Maxima5.5 (in GCL)    32.0 sec (? GC not reported, second trial)
Macsyma (commercial)  37.5 sec (including 29.78 in GC)
Maxima5.5 (in GCL)    53.5 sec (? GC not reported, first trial)
Maple7                50.0 sec (? GC, it was slower on 2nd trial)
Mathematica 4.1       82.3 sec (? it was slower on 2nd trial)
Form 3.0              97.9 sec (see notes)
MuPad 1.4.2 Pro      118.6 sec (general form)
```

We also collected some data for times on other machines:

```
Reduce 3.7* (PSL)      6.2 sec (including 0.22 GC)
Maxima 5.9.0(CMUCL)*   7.9 sec
Giac 0.2.2* (C++ prg) 13.5 sec
MapleVR3*             16.7 sec
MapleVR4*             17.9 sec
MapleVR5*             18.4 sec
Maxima 5.9.0(GCL-2)*  18.8 sec
Giac 0.2.2*           19.5 sec
Axiom 2.3*            20.1 sec (including 6.56 GC)
Maxima 5.9.0(clisp)* 31.30 sec
Maxima5.6*            51.0 sec
Ginac 1.0.8*          57.3 sec
Maxima 5.9.0(GCL-1)* 112.2 sec
Mupad 2.0*           117.0 sec
Mupad 1.4.2*         119.2 sec
Meditor*             204.5 sec
```

Notes on this table

*(Giac 0.2.2 compiled with gcc 2.96 on 800Mhz P-III 256K cache. Data provided by Parisse Bernard. Numerous tests were run by Richard Kreckel on a P-III 1GHz 512kB cache Linux machine. Times for his machine seems to be about the same as mine for MuPad 1.4.2 (general), Macsyma 5.6, Mathematica, so probably the numbers for Ginac-1.0.8, Mupad 2.0, MapleVR3,4,5 are about comparable. Meditor took 204.5 sec on a 700Mhz processor. Meditor run by Raphael Jolly.

The collection of Maxima 5.9 timings are from James Amundson, running on a 700Mhz P-III linux machine. The GCL-1 time is, we believe, for the same system that took 53 seconds on our machine. The GCL-2 time is given a larger initial memory allocation. Clisp is a byte-coded Common Lisp that one expects to be slower compared to direct instruction execution because it interposes an extra layer of interpretation. CMUCL is a highly optimizing implementation of Common Lisp based on work at Carnegie Mellon University. Currently CMUCL runs only on variations of the Unix operating system (Linux, Sun Solaris or similar). SBCL is a descendant of CMUCL and may eventually be available on other hosts. If it scales up comparably to the other measurements, it would be fastest of the Lisp systems. Axiom 2.3 time on Duron-600/Linux was is provided by Andrey Grozan. Reduce 3.7/PSL is from Andrey Grozin. PSL, Portable Standard Lisp, is a different version of Lisp, simpler than Common Lisp, and oriented toward support of Reduce.)

Sparseness

As an additional test we tried to probe the sensitivity of the programs to additional sparseness. In this benchmark we used $(1 + x^2 + y^2 + z^2)$ which makes (only) MockMMA substantially less space efficient: it uses twice as much storage for the polynomial "backbones" during multiplication and in the final result. This alteration makes the program run 12 percent slower, uses 16 percent more cons cells and 4 percent more "other" bytes which constitute bignums and vectors. There are 156 megabytes of such other material used during the computation. The Macsyma and Hashing code size and times do not change.

Isolating bignum costs, and effect of memory cache

Of these execution times, how much of it is attributable to bignum arithmetic, and therefore irreducible in the context of improving data structures?

We reproduced the timing with the same programs but with the change that all the coefficients in p^{20} were replaced with the integer 1. Thus the same number of arithmetic operations was still being done, but none of it grew to bignum size in the course of the calculation. We provide this data below, where we have also added some space data for the ACL runs.

```
 MockMMA (in ACL 6.1)  1.12 sec (including 0.12 in GC)
;  112,168 cons cells, 8,356,424 other bytes

 Macsyma (in ACL 6.1)  2.39 sec  (including 0.21 in GC)
;  3,174,312 cons cells, 96 other bytes

 Hashing (in ACL 6.1), 1.7 sec (including 0.02 in GC)
;  12,343 cons cells, 916,680 other bytes

Reduce 3.7 (in CSL)    3.6  (including 0.8 in GC)[Thanks to AC Norman]
```

Hashing is less local

Based on these measurements it would seem that in Macsyma the bignum arithmetic costs about 6.9-2.4 = 4.5 sec. For MockMMA, the time is 5.8-1.12 = 4.6 sec. For Reduce, the time is 5.0-3.6 = 1.4 sec. For Hashing, it seems the bignum arithmetic cost is about 7.9-1.7 = 6.2 sec. This peculiar result says that the *same arithmetic takes 1.7 seconds more when called from the Hashing program compared to being called from MockMMA*. Since the same instructions are executed on the same data, this requires some further study!

We believe the hashtable + bignum test runs slower because the input and output hashtables combined with the bignum routines, as well as the garbage collections prompted by the bignum arithmetic, in total cause more cache misses[4]. The data + keys of the two inputs is already several times the size of the L1 data cache. We must also figure that part of the output hashtable must also be present in the cache, which grows to contain

[4]Pentium 3 specifications: L1 cache, half data and half instruction, is 16Kbytes or 4K fullwords, the L2 cache is 256Kbytes

12,341 bignums and 12,000 keys). If there is too much data accessed in random fashion (especially so for the output hashtable), successive accesses to are likely to provoke L1 cache misses. But they are mostly within the L2 (256Kbyte) cache. When we move from the fixnum-sized coefficients to bignum coefficients we force rather numerous garbage collections. Switching between the processes of"mutation" and "collection" means the L1 cache gets spoiled (twice) at each GC. But we believe the more expensive problem may be the L2 cache miss rate which is raised by the many small "generation scavenging" GCs during this computation.

Cache performance also explains the relatively high speed of MockMMA, which uses vectors (recursively, of vectors, for each different variable). At the lowest level of the data structure we have a vector of coefficients which are bignums: as we pluck successive coefficients from the input to use as multipliers, the bignums and perhaps the lowest-level coefficient vector tends to persist in a cache.

Further tests on much larger problems than our benchmark here suggests that we are still well below the true "knee" of the cost curve on the L2 cache, which on this machine holds 256Kbytes. Even our 12,341-coefficient result fits in our L2 cache, and would work well if only the L2 cache were not being flushed from activities of the garbage collection (or for that matter, activities of the operating system.) From other tests on this same machine we would expect to suffer an additional slowdown of perhaps a factor of 7 if our data grew routinely to be much larger than the L2 cache [4].

Having seen the importance of locality of data in our programs we should also make some efforts to assure that the design of garbage collection algorithms and policies are cognizant of the issues. Numerous small GCs may be a disadvantage compared to an occasional large one since it will certainly flush the L1 cache and possibly some of the L2 cache. On the other hand the GC algorithm should not be delayed so long that the L2 cache will be filled and then overflowed from its activities.

We have not seen any previous report of Lisp using real-life cache miss data (that is, not just simulations), so there may be some novelty to this part of the report.

Lisp and cache

We have pursued such benchmarking in a different context separately[4], but briefly review the findings here. We used the PAPI system[5] adapted

[5]icl.cs.utk.edu/projects/papi/

to Lisp. We instrumented our system to count the total (instruction plus data) cache misses in the Pentium-3 L1 and L2 caches for various tests. All tabulated cache miss numbers should be multiplied by 10^7. For convenience we have also reproduced the associated run times.
HT=hash table version using bignums (the standard test)
MM= MockMMa for standard test
HT1 = hash table version with only 1's in the input

	L1 misses	L2 misses	runtime	cycles/L1 miss	cyc/L2miss
HT	4.15	2.79	7.9 sec	177	264
MM	2.62	1.45	5.8 sec	205	373
HT1	1.21	0.86	1.7 sec	131	184

A more detailed analysis would be required to assess the damage from the cache misses. In an attempt to rationalize the time difference between HT and HT1, assume that, *including some measure of related costs*, an L1 miss causes the waste of 12 cycles and an L2 miss similarly uses 120 cycles. Then we have lost $1.8 \cdot 10^9$ cycles or 1.92 seconds, accounting for most of the time difference between HT and MM. There are other memory related activities we have not accounted for, including TLB activity.

Blocking for cache performance

There is yet more that we can do, namely a block-multiply of polynomials, analogous to blocking of matrix operations.

A somewhat abstracted analysis of the cache performance of polynomial multiplication was done (for a class project) by Yozo Hida, in which he showed that performance can be improved by optimizing block sizes for multiplication. That is, if the two input polynomials are broken up into smaller sets of monomial terms, the "batched" results may be computed faster. A speedup of up to 47% was measured in this paper[5]. We will discuss further results along these lines in a future paper.

Our expectation is that programs based on linked lists would be able to take advantage of blocking for smaller size computations: the cache memory would be about half-filled with pointers. Thus there is a considerable advantage to "denser" representations. The same argument goes for bignums: the ACL bignum package uses only about half the bits of the words (to make multiplication "more portable"). Other systems (in particular GMP) using a 32×32 bit multiply have more compact bignum representations.

It is obvious from our tests that there are substantial variations in Lisp implementations in bignum efficiency that substantially affects the results. Certainly the Reduce CSL bignum arithmetic is more efficient than the competition since its *overall* time is lower than the *arithmetic-only* time for ACL. The pecking order for arithmetic is not entirely clear, but a separate comparison of CMUCL and ACL (on a Sun Solaris machine) suggests they are approximately equivalent on the "fixnum only" hash version, but CMUCL is about 11 percent faster when bignums are used.

All the formulations do the same coefficient arithmetic, essentially 3,136,441 multiplications and almost that many adds.

As indicated earlier, one way of overcoming a less efficient arithmetic system is to try GMP, loaded into ACL. We tested GMP versions 3.1 and 4.1, provided as libraries. An advantage of the 4.1 version, which is used both in our hashing program and in the MockMMA program, is a combined add-multipy call.

The result is that by using GMP, our total hashing time is much lower (4.7 rather than 7.9 seconds) for the size of coefficients in this benchmark. Similarly, the MockMMA benchmark dropped from 5.8 to 3.3 seconds, but there is a problem in that storage is used extravagantly. (That is, it is not collected properly, and grows rapidly.) A careful reprogramming of this recursive multiplication routine may help, but its functional orientation makes such a change very tricky. Using a variety of tricks to re-allocate GMP numbers by garbage collection or reference counts raises the time to about 9 seconds.

Using GMP more generally

From separate tests in ACL, we know that for raw arithmetic on much bigger numbers GMP is significantly faster: for squaring a number with 2^{17} bits GMP-3.1 is about 12 times faster, and GMP-4.1 is about 45 times faster.

For numbers with 2^6 bits (i.e. two-words long), ACL is twice as fast as GMP4.1 *if new space is allocated for each word.* GMP-4.1 is twice as fast as ACL if the same spot can be reused. At these rather small sizes the time for the arithmetic is dominated by the time to allocate space for a number (a necessary step if a bound on the number's length is not known until run-time.)

The key then, is for GMP to use an operation *not supported by Lisp or other typical functional programming systems* which is to modify a number *in place.* That is, GMP 3.1 allows one to compute `a:=a*b` and even better, `a:=a+b*c`, in GMP-4.1. The GMP advantage emerges since Lisp can provide essentially `newa:=a+b*c; a:=newa`. It then almost always discards the `newa` value. In this benchmark some 3,136,441 products and 3,124,100 sums are computed, but only 12,341 numbers survive. We have seen that collecting those discarded objects provokes garbage collections and cache misses.

Unfortunately a more realistic accounting of cost (which we have not accounted for in the GMP times above) requires us to compute when we can in fact return the GMP numbers to the storage in the GMP system, and actually returning them. The measured cost for the hashing version is small: to return all remaining 12,341 coefficients in this answer costs only 20 ms., *but this assumes we need do no computations to determine which GMP numbers are "garbage"*. It is much more complicated to deal with the storage in the other algorithms: In reality, using GMP as a substitute for all numbers in a Lisp system means we must separately collect garbage in this bignum space. We would miss the smooth transition managed by Lisp between single-precision and multiple-precision integers. Nevertheless, further work with GMP is possible and even has good prospects [3]. We have also programmed a half-page implementation of the Lisp GMP linkage where "finalizations" are attached to GMP object so that their internal storage can be returned when their "shell" is garbage-collected by Lisp. This keeps dead-storage from piling up, but itself adds storage overhead.

Can we change the benchmark to show GMP winning even more? Certainly: We increase the size of coefficients by starting with $10000000001(1+x+y+z)$. The coefficients in the answer are then about 10^{400}. Our tests show GMP-4.1 takes only 24 seconds (0.06 in GC), GMP-3.1 takes 149 seconds (0.06 in GC) while Lisp takes 167 seconds (21 in GC)[6].

Running the same modified test takes Pari out of first place: it takes 77 seconds, whereas Singular takes 28 seconds. Each of these is slower than GMP-4.1 Lisp. Of the 24 seconds, at least 11.5 are within GMP, 9.4 seconds within `gmp_add_n` alone.

Using GMP did not speed up our MockMMA version of polynomial multiplication as much as we had hoped because the program produces, recursively, many sub-polynomials which are then added together and should be discarded. This creation of many numbers doesn't make good use of GMP-like facilities. We think a substantial revision of the MockMMA version in which the basic operations are "imperative" rather than functional, may be faster. That is, the polynomial operations would be based on in-place alterations.

[6]Lisp takes 145 seconds if the coefficient multiplier is 10000000000. It appears some optimization in Lisp takes advantage of words of zeros!

Quibbles about the answers, and a few minor points

Arguably the Hashing version is returning less information in the answer: the monomials are not sorted, and thus such information as "what is the highest degree term in z?" cannot be answered without some search. This kind of representation is also used in Maple, and we have argued previously that to be fair in comparisons, we should insist that Maple sort its final answers. In our example, a final sorting of the hash-table is a small fraction of a second.

Another quibble, perhaps more of a concern, is that the cost for Hashing will increase if the exponents cannot be encoded in a single word, or if we use a short vector of exponents as a key: the exponent arithmetic becomes more costly. The MockMMA version already allows such generality and is therefore another recommendation for it.

An anecdote about FORM

One of our initial tests suggested that FORM could compute the answer very fast: in about 0.9 seconds. But then a minor variation on that computation took 98 seconds. After an exchange of messages with Jos Vermaseren, we looked at the benchmark more carefully. Apparently FORM "expanded brackets" differently from the other programs, and instead of computing $p^{20} \cdot (1 + p^{20})$ it computed the equivalent $p^{20} + p^{40}$, a much quicker computation on any of the systems. It appeared that the 98 second time was more representative of the computation. Starting from this computation and following a suggestion by Jos Vermaseren to insert "sort" directives, we made FORM much faster, reducing the time from 98 to 22 seconds.

Conclusions

A computer algebra system (CAS) needs a multiplication program that is efficient on a typical mix of inputs. Not every CAS expression is a polynomial, but by suitable renaming of indeterminates many computations, we can reduce many bulky and time-consuming calculations to operations on polynomials. At the same time we are addressing multiplication, we presume the same underlying representation will support other operations, most particularly addition, subtraction and division with remainder. Our idea is to allow arbitrary size integer coefficients, any number of variables, and their exponents could be any positive integers.

However, we have made some judicious simplifications. For example, do we really need to represent a polynomial whose degree exceeds the maximum 32-bit signed integer? While some computer algebra systems allow this, we might not actually find this useful and would resent any substantial inefficiency this might force upon us. Do we need a system with 80,000 different variables?

Given the context of our tests, the major conclusion of this paper is: *memory access speed critically affects computation time for sparse polynomial multiplication.* Hashtable organization which might ordinarily seem fruitful is not as good as expected, since at this size of computation it forces a less effective use of memory cache.

Our original intention at the outset was to see if we could write a compact program to do sparse polynomial multiplication faster than typical systems, but with comparable generality. Our hope was that the hash-table mechanism provided in Common Lisp would enable this. An exceedingly short first program computed the answer, but improving its efficiency made it somewhat larger in size.

It also became apparent that systems with many short garbage collections are at a disadvantage in a cache-memory system since the alternation between computation and collection forced cache misses. Tuning of a garbage collector to take larger chunks of time less frequently should provide better performance. In our tests, using bignums forced frequent GC interruptions. This had more of an effect on the hash program, which already had less locality of data and hence took more time to move into the cache.

Writing in Lisp, a functional language in which `x := x+a*b` requires creating a new value and "pointing" `x` to it may be slower in certain applications like this than an alternative version of bignum arithmetic with "in-place" non-functional style operations. (That is, using unit operations that assign `x := x+a*b` destroying the old value of `x`).

When we expanded the scope of this paper to see how other systems behaved, we found that Maple 7, Mathematica, Mupad and FORM are not nearly as fast as we had expected, but that GP/Pari, Reduce and Singular made good showings.

Thanks to Richard Kreckel, we learned that earlier versions of Maple were much faster on this benchmark.

Given a constant Lisp environment (Allegro Common Lisp), MockMMA's mixture of dense arrays and recursion seems to work well for this problem. Hashing (using "containers" to avoid re-hashing) is better than lists but slower than arrays. If all coefficient arithmetic is done in fixed-size chunks, the compactness of the hash program may be advantageous.

Using different Lisps and different initial allocations of working storage provides substantially different apparent performance. CSL, which is not a full Common Lisp, but a Lisp-based system building tool, provided an especially well-tuned base for Reduce. An important contributor to the speed of this system is its bignum arithmetic which was about 3 times faster. GCL and CLOE Lisp based Macsymas were apparently hobbled by inadequate storage allocations and/or less efficient garbage collection algorithms as initially configured.

Fermat, a relatively smaller interactive system tuned for performance is respectable but was not a top contender on this benchmark straight out of the box. Upon seeing his program's rank on this test, R. Lewis, its author, looked at his algorithm and realized that the "efficient" algorithm he used, a Karatsuba-style polynomial multiplication, was actually slower than the naive algorithm. Forcing the use of the naive algorithm sped up the system from 8.6 to 4.5 seconds. We put the shorter speed on the chart with the notation "see notes".

We anticipate that faster bignum arithmetic (for example, from GP/Pari) could be incorporated into any of the slower Lisp systems. In addition to GP/Pari's arithmetic it may be worthwhile to continue to follow the revisions of GMP, as well as David Bailey's MPFUN to see which off-the-shelf bignum library might make a net improvement on this benchmark. Such a library system, even if it is fast, still needs to be neatly merged into Lisp including techniques for managing storage. MPFUN packaged as a windows dynamic library can in principle be loaded into our standard Lisp environment, just as we have done with GMP.

Finally, we wish to point out that in May, 2002 there are at least 12 computer algebra programs that can be run on a Windows/Intel computer, each competent to do multivariate polynomial multiplication where there are over 10,000 terms in the answer.

Acknowledgments

This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley. Thanks to the contributors of additional timings who responded to a post on the `sci.math.symbolic` newsgroup, and of course thanks to the builders of the computer algebra systems tested above. Thanks also to Duane Rettig and John Foderaro for help on simplifying the interface between Allegro Common Lisp and GMP. Thanks to John Chen and Idler Fang for assistance with obtaining the GMP-4.1.dll. Thanks also to Ronen Gradwohl who has assisted in performing some of the experiments, and Derek Wollenstein for additional comments.

Program listings and notes.

(online at then end of `http://www.cs.berkeley.edu/~fateman/papers/fastmult.tex` or available from author.)

References

[1] A. D. Hall. “The ALTRAN System for Rational Function Manipulation - A Survey”, *CACM 14(8)*:517-521 (Aug 1971).

[2] R. Fateman. “A Lisp-language Mathematica-to-Lisp Translator,” *SIGSAM Bulletin 24* no. 2 p 19-21 (April, 1990). Also reprinted in *Computer Algebra Nederland Nieuwsbrief 6*, October, 1990.

[3] R. Fateman. “Importing Pre-packaged Software into Lisp: Experience with Arbitrary-Precision Floating-Point Numbers,” Poster session, ISSAC 2000 International Symposium on Symbolic and Algebraic Computation, St. Andrews, Scotland, UK, August 2000., `http://www.cs.berkeley.edu/~fateman/papers/mpflis.pdf`

[4] R. Fateman. “Memory Cache and Lisp: Faster list processing via automatically rearranging memory,” Draft, May 2002.

[5] Yozo Hida. “Data Structures and Cache Behavior of Sparse Polynomial Multiplication,” Class project CS282, UCB May, 2002.