Dumb = Fast for Polynomial Multiplication. Or, You have Plenty of Memory. Use Some.

Richard Fateman

January 25, 2013

Abstract

We've read (and written) papers on how to write programs for multiplying polynomials, but a simple fact seems to have eluded some researchers, and we have not really emphasized it enough. Here it is: By using some extra memory you can simplify the program so much that it is very fast and hard to beat with any more sophisticated method. We explain why a very simple program is so fast that it probably should be favored over more sophisticated routines much of the time.

1 The task

Abstractly we can consider a polynomial as a list or array of coefficient-exponent pairs. For example, $7 + 9x + 34x^5$ might be `((0 . 7)(1 . 9)(5 . 34))`. Another way (which appears to be sometimes slightly preferable for some technical reasons having to do with space and speed of access) is to use two arrays, `(0 1 5), (7 9 34)`.

For concreteness we have decided that the sequences will be in ascending order of exponent. The multiplication program returns a new polynomial which is the product of the two inputs, in the same form. Although this seems to work for only univariate polynomials, we know that polynomials in several variables can be mapped into univariate polynomials, by packing exponents with respect to different variables together in single exponents, and so we do not discuss this further.

2 An example and an algorithm

Say the polynomial is quite sparse, say $7 + 9x^{103} + 34x^{200}$ and for simplicity of description we multiply it by itself. This gives a polynomial of degree 400 of size 9. (It has 9 non-zero terms.)

Here is one way to do this. Program MUL-DENSE. (Yes, we are multiplying sparse polynomials, but we are using a method that would seem more appropriate for dense polynomials) (already described in [1], and fairly obvious):

1. Allocate an array A of size sufficient for degree of answer (here 401),
2. Fill A with zeros.
3. Compute the product terms and accumulate as necessary, and insert the products into A. (For this example there are 9 terms)
4. Traverse A and collect the non-zero terms into two arrays of minimum size. (For this example, each of size 9) the exponents E and coefficients C.
5. Return the pair (E C).

Any of a large number of clever algorithms can be used for step 3, especially if A is some alternative data structure and mechanism for storing and retrieving integer-indexed data. However, assuming an array is in fact used, any algorithm's running time depends on the degree d of the answer in steps 1, 2, and 4, though not necessarily dominated by these times. These steps are linear-time in d[1]. Step 3, if a simple algorithm computing the cross-product is used, depends only on the product of the number of non-zero terms (or size) of of the inputs. Thus for this step, $7 + 9x^{10003} + 34x^{20000}$ squared runs just as fast.[4, 1][2].

In benchmarks for this task at degree 401, MUL-DENSE is twice as fast as a heap-based stand-alone program provided by Roman Pearce, implemented according to the design he advocates [4]. *This test shows that, even though only 3 out of 401 coefficients are non-zero, it's faster to just squander 99 percent of the memory.* The advantage grows if the polynomial is just a little denser even if it is obviously not truly dense: If we pack the 3 non-zero coefficients into a polynomial of smaller degree, say 16, MUL-DENSE is perhaps 12 times faster.

Where does this simple-minded program spend most of its time?

We consider again the costs in step 1, 2, and 4. We cannot do much to speed up steps 1 and 2, which are not so expensive anyway compared to 4, but they are all, as noted, linear in the degree of the answer.

Essentially, we compare it to other programs that are also based on essentially the same step 3, computing P*Q by doing size(P)*size(Q) multiplies[3].

We could try to make the steps 1,2,4 run faster, by for example being clever in designing looping constructs to avoid memory cache misses [4], but our examples are not so huge. We could use a cleverer storage structure that is not O(degree) but O(size) like a hash table, or one that also preserves order while inserting, like a skip list, or one of the many variants of trees or a heap [4], and thereby reduces or eliminates step 4.

Intuition suggests two situations in which these alternative ideas might off. Huge huge size (relative to memory cache) or small size but absurdly sparse polynomials: huge huge degree. Experience suggests these don't occur, generally. We return to these situations in a later section and show an additional design idea.

For now we concentrate on the probable realm of practical computations in a computer algebra system. Let us say that in a particular run, MUL-DENSE uses 50 percent of its time in steps 1,2,4 (reminder: this consists of allocating the array and later scanning for non-zero terms).

It is unlikely that you will be able to write any program that will do the same overall multiplication twice as fast. Why?[5] Even if you zeroed out the cost of steps 1,2,4, you would make the whole algorithm just twice as fast, leaving any further improvement to reducing the number of (or cost of) multiplications and adds. For sparse inputs it is hard to find a way to do less work than MUL-DENSE in step 3, which here is essentially a double-loop over A[i]:=A[i]+B[j]*C.

Indeed, since a typical run might spend 10 percent of its time in step 1,2,4, concentrating on this data-allocation aspect would be unlikely to beat MUL-DENSE by even 10 percent.

In a somewhat more dense situation, the answer array A may be mostly filled even if the inputs are not especially dense, and in MUL-DENSE all of the cost is expended in step 3, making it hard to speed up the program (we have not found an implementation to do this, though Monagan and Pearce [4] claim that by somewhat delaying the arithmetic, at least if it is expensive "bignum" computation, they can achieve a faster throughput.) But in our implementation it is certainly difficult in this circumstance to speed up MUL-DENSE by storing terms in some alternative to an array, unless that alternative is cache-efficient and the array is not.

MUL-DENSE, written to be a "good for mostly dense" algorithm surprised us when we first tested it on sparse cases. It is pretty good even for rather sparse polynomials, say where 98 percent of the possible coefficients are zero. For extremely sparse polynomials, it is possible to segment the inputs into sections of relatively dense polynomials, and use another algorithm for assembling the collection of cross-multiplications

[1]Strictly speaking, only if memory access is uniform. If the array is so large that the array A is splayed out to pages on disk, additional costs are involved.

[2]Assuming the exponents are ordinary numbers and not themselves "bignums" on which the operation of addition requires many hardware operations.

[3]If we can quickly recognize,*a priori*, that the task consist of multiplying substantially, if not entirely, dense inputs of large size, we should mention that there are asymptotically faster versions being based on the Fast Fourier Transform. (Fateman [1] among many others provides further discussion and references.) Most implementations of an FFT will be oblivious to the existence of zero entries. Considerations of how "large" must be for this to be advantageous depend on strongly on implementation considerations.

[4]For large enough inputs with simple-enough coefficients, Hida [3] shows examples where this can be sped up by perhaps 47% by "blocking" to improve cache performance. Such improvement is likely only if the coefficient operations themselves are simple (e.g. floating-point) and do not require arbitrary-precision arithmetic, following pointers etc.

[5]Amdahl's law

using MUL-DENSE. This collective segmented MUL-DENSE algorithm may or may not be advantageous compared to other algorithms, and the collection can be done in various ways [5]. The key really is to rapidly choose the best program based on some easily computable criteria. Since the time taken for polynomial multiplication is closely correlated with the sparseness of the answer, and this is not easily predicted from the sparseness of the inputs, we generally need to make an approximate decision. Finding a really good rapid assessment of the situation is a puzzle, though it need not be completely resolved. One aspect is the availability of an easy partial decision. If MUL-DENSE is say 12 times faster than the next best on easily identified common tasks, these tasks should be shipped off to MUL-DENSE. It doesn't have to be particularly dense or small for this program to win. In fact, in current computing situations, it is quite typical to have a few gigabytes of memory (RAM) not in use, and a few megabytes of medium or high-speed cache not in high demand, and MUL-DENSE may make good use of this. Consider a large example: P, Q each a polynomial of degree 30,000, stored explicitly with exponents and coefficients each 4 bytes: 240,000 bytes. The product takes another 480,000 bytes. So this problem fits in one megabyte, easily in L2 (my desktop computer's) cache. Say that P and Q are of degree 30,000, but only 3,000 or even 300 of the coefficients are non-zero. No problem. Given the sizes of contemporary caches, we could easily multiply these sizes by 4; if we are concerned mostly with RAM not cache, we can multiply them by 1,000. Note that computer algebra systems would not usually be required to operate on such large examples; such polynomials typically are encountered only in artificial benchmarks.

3 Asymptotic analysis misleads

While we hesitate to dismiss entirely an analysis for choosing an algorithm in terms of cache performance, it seems likely that for many cases automatic cache management and a large enough cache will keep what is needed readily accessible. In other words, ordinary cache management is likely to work fine.

Methods for making optimal decisions for far larger and far sparser problems, (should they ever occur in practice) is not so clear-cut, but see the analysis by Roche [5].

As Roche says in the context of his improved "Chunky" multiplication, "An important requirement, however, is that the worst-case complexity still matches the best-known algorithms, resulting in routines which are often faster but never asymptotically slower than traditional algorithms."

While worst-case complexity and asymptotic speed are fine for writing theorems, I would emphasize (and I think Roche agrees) that the proof is in the pudding: For evaluating a program for incorporation into a computer algebra system, the goal is that the measured performance is comparable to the best measured performance of any available algorithm for any inputs, not just artificially large ones. Benchmarking of programs implementing "asymptotically fast" algorithms demonstrate that such programs can be disappointingly slow on inputs of realistic size.

4 Further design experience

One *can* create examples where MUL-DENSE is forced to allocate an astronomical-sized array, even if the answer is small, and so it *can be beaten* on some examples. There are counter-strategies though, and we have programmed some of them[6].

One way to defend MUL-DENSE from an adversarial super-sparse case is to modify it slightly and produce a more sophisticated version that first checks for the case of "excessive" sparseness. We call this MUL-PA (PA= polyalgorithm). MUL-PA looks for long gaps and can therefore multiply a polynomial r by a polynomial $px^{1000} + q$ by computing rp and rq separately and returning $rpx^{1000} + rq$. This segmentation can be repeated recursiv ely if p, q or r have large gaps.

This points out the direction we see as most appropriate overall, which is to spend a bit of time analyzing the task at hand. We hope that small problems can be recognized and dispensed with rather quickly. It may pay to analyze larger problems in (at least) the following respects:

[6]Testing against reality encounters the difficult of guessing at "real" cases.

1. If the coefficient domain is such that all coefficients are guaranteed to be single-word fixed integers, consider specializing the data allocation type of the array A, and the operations on it. This would be not only the case of a finite-field of less-than-word size, but polynomials with small coefficients. One "poly-algoritm" we have written computes a norm of the input polynomials so that one can bound the coefficients in the product.

2. Similarly if the coefficients are one of the floating-point types, a float-array can be allocated, with float operations. Some Lisp systems support single, double, double-double float types.

3. The alternative of the standard computer-algebra domain of arbitrary-precision integers is more general and costly, but even in this case there may be a choice to make. Based on the coefficient bound computed earlier, it may be advisable to use an external library such as GMP or MPFR to perform the arithmetic. Some Lisp systems make their own tradeoff between "internal" bignums and operations handed off to GMP or other external libraries.

4. If there is an especially large discrepancy between the degree and the number of terms then there may be one or more opportunities for gap analysis as indicated above. Some heuristics for choosing gaps may be appropriate.

5. (More drastically) A polynomial product may be stored in factored form, assuming that the explicit coefficients are not necessary immediately. The actual multiplication may be deferred, until after (say) further multiplications and divisions. As a consequence of cancellations the multiplication may be totally avoided.

5 Conclusion

Sometimes a simple program is fast. No amount of fancy analysis will get around the fact that the simple program should be used most of the time.

References

[1] Richard Fateman. DRAFT 11: What's it worth to write a short program for polynomial multiplication?, Online at `http://www.cs.berkeley.edu/~fateman/papers/shortprog.pdf`

[2] Stephen C. Johnson. Sparse polynomial arithmetic. SIGSAM Bull., 8(3):63–71, 1974. ISSN 0163-5824. doi: http://doi.acm.org/10.1145/1086837.1086847.

[3] Yozo Hida, "Data Structures and Cache Behavior of Sparse Polynomial Multiplication," Class project CS282, UC Berkeley, May, 2002. `http://www.cs.berkeley.edu/~fateman/papers/yozocache.pdf`

[4] Michael Monagan, Roman Pearce. "Polynomial Division Using Dynamic Arrays, Heaps, and Packed Exponent Vectors" in *Computer Algebra in Scientific Computing*, Lecture Notes in Computer Science Volume 4770/2007, 2007 295–315

[5] Daniel Roche. Chunky and Equal-Spaced Polynomial Multiplication (PDF). Submitted to Journal of Symbolic Computation, November 2008. Online at `http://arxiv.org/abs/1004.4641`