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

Richard Fateman

December 8, 2010

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 too fast 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 univariate polynomials, we know that polynomials in several variables can be mapped into univariate polynomials, 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 terms.)

Here is one way to do this. Program MUL-DENSE. (already described in [1], and fairly obvious):

1. Allocate an array A of size 401,
2. Fill A with zeros.
3. Compute and accumulate as necessary, and insert the 9 products into A.
4. Traverse A and collect the non-zero terms into two arrays. 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 if A is some alternative data structure for a smart array or tree or hash table. Indeed it can be any mechanism for storing and retrieving indexed. The algorithms' running time apparently depends on the degree of the answer in steps 1, maybe 2, and 4.

Linearly on the degree of the answer[1]. Step 3 depends only on the number of non-zero terms, or size. 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 suggests 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 somewhat denser even if it is 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? As we use it for yet sparser and sparser situations, 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.

It seems necessary to make a fairly strong statement here regarding why this program, for most inputs, is going to be fast relative to any other that is based on computing P*Q by doing size(P)*size(Q) multiplies[3].

How can we make programs that compute the same result run faster? We could try to make the same steps run faster, by for example being clever in designing looping constructs to avoid memory cache misses [4]. We could use a cleverer storage structure that is not O(degree) in size like a hash table, or one that also preserves order while inserting, like a tree or a heap, and thereby reduce or eliminate step 4.

We can see only two situations in which these alternative ideas pay off. Huge huge size (relative to memory cache) or small size but absurdly sparse polynomials: huge huge degree.

Let us remove these two situations from consideration, which leaves us in the realm of probably most practical computations in a computer algebra system. Let us say that in a particular run, MUL-DENSE uses 50 percent of its time in step 4. It is unlikely that you will be able to write any program that will do the same task twice as fast. Why? 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.

In this circumstance, even if you entirely eliminate step 4, you would get a factor of two. Not more. Indeed, since a typical run might spend 10 percent of its time in step 4, you would be unlikely to beat MUL-DENSE by even 10 percent.

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 sparse examples. One way to defend MUL-DENSE is to write a slightly more sophisticated version that checks for the case of excessive sparseness; one approach we have written basically uses MUL-DENSE on clumps, allowing for long gaps to be ignored. Another solution is to use a substantially different approach, or to use the same algorithm but with another data structure (say a hash-table or a heap rather than an array).

In a somewhat more dense situation, the array A will 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 (not impossible– Monagan and Pearce claim that by somewhat delaying the arithmetic, at least if it is expensive "bignum" computation, one can achieve a faster throughput.) But it is certainly difficult in this circumstance to speed up MUL-DENSE by storing terms in some alternative to an array.

MUL-DENSE, written to be a "good for mostly dense" algorithm 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 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 pre-computable

[1]Linear in time only if memory access is uniform. If the array is so large that the array A is splayed out to pages on disk, there is a significant penalty.

[2]Assuming the exponents are ordinarynumbers and not themselves "bignums" on which basic operations like addition require many hardware operations.

[3]If we can quickly recognize,*a priori*, that the task consist of multiplying substantially, if not entirely dense inputs of substantial size, there are other methods that are plausible, the asymptotically best being based on the Fast Fourier Transform. (Fateman [1] among many others provides further discussion and references.) The FFT is oblivious to the existence of zero entries.

[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.

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. A good rapid assessment of the situation represents a current difficulty.

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 gigabyates of memory (RAM) to spare, and a few megabytes of medium or high-speed cache, 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 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 of choice for this 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.

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 actual performance is comparable to the best performance of any available algorithm for any inputs, not just large ones. Benchmarking of programs implementing "asymptotically fast" algorithms demonstrate that such programs can be disappointingly slow on inputs of small, and even large but "finite" size.

4 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