J. Symbolic Computation (1991) **11**, 307-320

Parallel Multiplication and Powering of Polynomials

CARL G. PONDER

Computer Science Division, University of California, Berkeley, CA 94720, *USA*

(*Received* 24 *March* 1988)

This paper examines the most efficient known serial and parallel algorithms for multiplying and powering polynomials. For sparse polynomials the *Simp* algorithm multiplies using a simple divide and conquer approach, and the *NOMC* algorithm computes powers using a multinomial expansion. For dense polynomials the *FFT* multiplies and powers by evaluating polynomials at a set of points, performing pointwise multiplication or powering, and interpolating a polynomial through the results. Practical issues of applying these algorithms in algebraic manipulation systems are discussed.

1. Introduction

Polynomials represent an important class of expressions in algebraic manipulation. Efficient operations on polynomials are requisite for an efficient algebraic manipulation system. In designing a parallel system for algebraic manipulation, using parallelism in performing polynomial operations is one of the more attractive techniques to keep processors computing productively during substantial amounts of computation.

Efficient serial algorithms exist for multiplying and powering sparse polynomials. The *FFT* algorithm for multiplying and powering dense polynomials is the most efficient known, demonstrating superiority to other algorithms (such as *Eval* (Fateman, 1974*a*; Knuth, 1969) and *Karatsuba* (Alagar & Probst, 1987; Fateman, 1974*a*; Knuth, 1969) both in asymptotic analysis and (except for very small cases) empirical evaluation. It has been conjectured to be asymptotically optimal.

It is the purpose of this paper to explore the potential of these algorithms for parallel execution. We assume a model of parallel execution based on low overhead shared-memory multiprocessing. Some simple empirical studies for up to four processors suggest these methods work. The data structures used to represent polynomials have some effect on the efficiency of the algorithms.

2. Characteristics of Polynomials

We will refer to the number of non-zero monomial terms in a polynomial p as size(p). We compute degree(p) (the *total degree of* p) by summing the exponents in each monomial, and taking the maximum of these sums. This definition of *total degree* is used in papers on Gröbner-basis reduction (Buchberger, 1985).

Alagar & Probst (1987) define the term *uniformly dense* to describe a multivariate polynomial whose size is nearly maximal for the given total degree. An example is the expansion $(x+y+1)^n$ which has the form

$$1+nx+ny+\tfrac{1}{2}n(n-1)xy+\cdots$$

and includes all products of powers of x and y up to total degree n. The complementary term *non-uniformly dense* is used to describe a polynomial such that each variable appears raised to nearly every degree, but the number of terms is not maximal. An example is the convolution

$$x^d y^0 + x^{d-1} y^1 + \cdots + x^0 y^d$$

where only terms of total degree d are present, though the degrees of x and y range from 0 to d. A polynomial is *sparse* if it has few terms relative to the maximum possible for the given total degree. The convolution is an example of a polynomial which is sparse and non-uniformly dense at the same time. A polynomial cannot be both sparse and uniformly dense. Obviously, *dense* and *sparse* are qualitative terms. Some algorithms are very efficient for dense polynomials but highly inefficient for sparse polynomials.

The following relations show how degree and size are bounded:

$$\text{degree}(p_1p_2) = \text{degree}(p_1) + \text{degree}(p_2) \tag{1}$$

$$\text{size}(p_1p_2) \leq \text{size}(p_1) \cdot \text{size}(p_2) \tag{2}$$

$$\text{degree}(p^k) = k \cdot \text{degree}(p) \tag{3}$$

$$\text{size}(p^k) \leq \binom{\text{size}(p)+k-1}{k}. \tag{4}$$

The maximum size of a polynomial for a given total degree is given by the relation

$$\text{size}(p) \leq \sum_{i=0}^{v} \binom{v}{i} \binom{\text{degree}(p)}{i}, \tag{5}$$

where v is the number of variables. For univariate polynomials $v = 1$ so this reduces to

$$\text{size}(p) \leq \text{degree}(p) + 1. \tag{6}$$

For v variables where exponents run from 0 to n in each variable,

$$\text{size}(p) = (n+1)^v. \tag{7}$$

Over a domain with zero-divisors, relation (1) is adjusted to

$$\text{degree}(p_1p_2) \leq \text{degree}(p_1) + \text{degree}(p_2). \tag{1a}$$

Relation (4) is derived as follows (Fateman, 1974*b*): let $p = A + B$, where B is a monomial and A contains all remaining terms. Then

$$p^k = (A+B)^k = \sum_{i=0}^{k} \binom{k}{i} A^i B^{k-i}.$$

If no collapsing occurs, as happens for suitably sparse p, each $A^i B^{k-i}$ pair will contribute exactly $\text{size}(A^i) = \text{size}(p) - 1$ terms. If A is a monomial this degenerates to one term per pair, or $k+1$ terms. Otherwise we have a total number of terms

$$\begin{aligned}\text{size}((A+B)^k) &= \sum_{i=0}^{k} \text{size}(A^i) = \sum_{i=0}^{k} \binom{(\text{size}(p)-1)+i-1}{i} \\ &= \sum_{i=0}^{k} \binom{\text{size}(p)+i-2}{i} = \sum_{i=0}^{k} \binom{\text{size}(p)+i-2}{\text{size}(p)-2} \\ &= \binom{\text{size}(p)+k-1}{k}\end{aligned}$$

using the identity

$$\sum_{i=0}^{k} \binom{r+i}{r} = \binom{r+k+1}{k}.$$

The size for sparse polynomials will grow at most quadratically as multiplications are performed. The degree will grow at most linearly under both multiplication and powering. Since size is ultimately bounded by degree, quadratic growth cannot be sustained under repeated multiplications as polynomials "fill in". As the "density" of the results increases, relatively fewer distinct terms will be generated by multiplication. For completely dense univariate polynomials, the size and degree grow linearly under multiplication and powering. The fastest the size of p^k can grow is approximated by

$$\binom{\text{size}(p)+k-1}{k} \sim \frac{\text{size}(p)^k}{k!} + O\left(\frac{\text{size}(p)^{k-1}}{2(k-2)!}\right)$$

for large k and increasing size(p).

3. Complexities of Multiplication and Powering

The complexity measures we concern ourselves with are the number of coefficient additions and multiplications, and the number of exponent comparison steps required to order the result. We will consider any of these to be "scalar" although coefficient operations may be floating point or arbitrary precision, and hence are potentially more expensive than exponent comparisons. We are also concerned with the maximum number of processors that can be kept busy with useful work in parallel algorithms. The parallel complexity is the maximum number of scalar operations used by any parallel branch of the computation. The algorithms we present are "balanced" in the sense that we attempt to "farm out" all parallel computations to processors in equal "chunks". The maximum and average number of scalar operations required per processor are close. For an ideal algorithm, the processor-time product, a good measure of parallel efficiency, is the same as the serial complexity.

Coefficient multiplication is probably the most expensive of the scalar operations as used in the Macsyma rational function package (Fateman, 1979). For sparse polynomials the parallelized algorithms we look at tend to parallelize efficiently the multiplication operations, but require serialized exponent addition or comparison to combine separate subresults. The exponent operation count does not dominate serial computation for reasonable-size input, and we conjecture that exponent operations will become only slightly more important in parallel.

We begin by showing that the number of scalar multiplications required for polynomial multiplication or powering depends strongly upon the size of the result.

LEMMA 1. *At least* size(p_1p_2) − size(p_1) − size(p_2) *multiplications are required in the worst case to compute* p_1p_2.

PROOF. Let A, B, and C be the sets of coefficients in p_1, p_2, and p_1p_2, respectively. If the elements of A and B are algebraically independent, the elements of C will be (at least) linearly independent. Given that we have generated C with h multiplications, let

$E = \{e_1, \ldots, e_h\}$ be the set of products produced by the multiplications. C must be formed by linear combinations of A, B, and E:

$$c_j = \sum_{i=1}^{h} x_{i,j}e_i + \sum_{i=1}^{n_1} y_{i,j}a_i + \sum_{i=1}^{n_2} z_{i,j}b_i$$

$$1 \le j \le \text{size}(p_1p_2).$$

Since the elements of C are linearly independent, it must be the case that $\text{size}(p_1p_2) \le h + \text{size}(p_1) + \text{size}(p_2)$, or $h \ge \text{size}(p_1p_2) - \text{size}(p_1) - \text{size}(p_2)$.

As a simple corollary, for p_1 and p_2 "sufficiently sparse", $\text{size}(p_1p_2) = \text{size}(p_1) \cdot \text{size}(p_2)$, so roughly $\text{size}(p_1) \cdot \text{size}(p_2)$ scalar multiplications are required. The *Simp* algorithm operates in exactly this bound. For "sufficiently dense" polynomials, $\text{size}(p_1p_2) = \text{size}(p_1) + \text{size}(p_2)$ so the lower bound on multiplications is linear. The best known algorithm in this range of densities is the *FFT* algorithm, which uses $\Theta((\text{degree}(p_1) + \text{degree}(p_2)) \cdot \log(\text{degree}(p_1) + \text{degree}(p_2)))$ scalar multiplications and is conjectured to be optimal in this regard. For powering, we have the similar result (Fateman, 1974*b*):

LEMMA 2. *At least* $\text{size}(p^k) - \text{size}(p)$ *scalar multiplications are required to compute* p^k.

PROOF. As before, the coefficient set C is linearly independent so the number h of intermediate products must satisfy the inequality $\text{size}(p^k) \le h + \text{size}(p)$, or $h \ge \text{size}(p^k) - \text{size}(p)$.

We know from relation (4) how the size of p^k can grow for sparse polynomials. The algorithm *NOMC* operates asymptotically with this. For dense univariate polynomials the growth is bounded more strictly by relations (3) and (5), giving a lower bound of $k \cdot \text{size}(p)$. Again, the best known algorithm for this case is the *FFT* using $\Theta(k \cdot \text{size}(p) \cdot \log(k \cdot \text{size}(p)))$ scalar multiplications and is likewise conjectured to be optimal in this regard.

For sparse polynomials, comparison operations are required to order the results. The problem of ordering the monomials of the product $p_1 \cdot p_2$ is equivalent to the "Sorting X+Y" problem (discussed by Harper *et al.*, 1975) of ordering all pairwise sums of the (ordered) elements of two vectors. For $\text{size}(p_1) = \text{size}(p_2) = n$, the lower bound on the number of comparisons is $\Omega(n \lg n)$. A solution by Jean Vuillemin (pers. comm.) uses $O(n^2)$ comparisons, i.e. proportional to the largest possible size of the result. Ordering the monomials of p^k is equivalent to sorting all distinct k-wise sums of elements of the vector X. Vuillemin's result is extended by recursively sorting the sets of $\lfloor k/2 \rfloor$- and $\lceil k/2 \rceil$-wise sums and sorting the pairwise sums of these two sets as before. The work to sort the final set will dominate the lower-order sets (see the analysis of *NOMC* in section (5.2)), so the total number of comparisons used is proportional to the largest possible size of the result

$$\binom{\text{size}(p) + k - 1}{k}.$$

Terms are combined by the addition operations, once comparisons have established that they are additive. Thus a lower bound on the number of coefficient additions required is

$$\text{size}(p_1) \cdot \text{size}(p_2) - \text{size}(p_1p_2)$$

or

$$\binom{\text{size}(p)+k-1}{k}-\text{size}(p^k),$$

which is the number of terms possible under relations (2) or (4), minus the actual size of the result. In the worst case $p=p_1=p_2$ and are completely dense. Under relations (1), (3), and (6) the number of coefficient additions required are

$$\text{size}(p)^2-2\cdot\text{size}(p)$$

or

$$\binom{\text{size}(p)+k-1}{k}-k\cdot\text{size}(p)\sim\frac{\text{size}(p)^k}{k!}-k\cdot\text{size}(p)$$

for multiplication and powering, respectively. Assuming that the results are ordered, the coefficient additions amortize into the exponent comparisons. The addition operations involved in generating new exponents amortize into the coefficient multiplications.

It is not obvious how to parallelize the addition or comparison operations in either the *Simp* or *NOMC* algorithms. In each case terms generated by different processes may add together (or even cancel). In both the *Simp* and *NOMC* algorithms we will use a parallel mergesort (Aho *et al.*, 1974) (just parallelize the recursive calls), which requires

$$\Theta\left(x+\frac{x}{y}\log\frac{x}{y}\right)$$

comparisons to sort x items with y processors, y ranging from 1 to x. For fixed y and increasing x it approaches a speedup linear in y, though this is not a linear speed-up over Vuillemin's result. A reasonably efficient parallel mergesort has been developed by Cole (1986), which takes $\Theta(\log x)$ operations to sort x items with x processors. Parallelism is applied to operations on individual elements, a very fine level of granularity. This process still appears inefficient and unnecessarily complicated for (small) fixed numbers of processors and large inputs, particularly if there is a significant overhead to inter-processor communication or shared memory access.

A hash table of monomials can be used as an unordered representation of polynomials (Goto & Kanada, 1976). A hash table can be updated in parallel, or used in serial to reduce the number of comparisons required to combine terms. The integrity of a hash table is difficult to maintain under parallel updates. If this were not an issue, the operations involved with combining terms would parallelize perfectly. The details involved with locking the hash buckets are complicated enough to possibly negate any advantages. Additional overheads such as computing the hash function are also significant.

4. Operations on Dense Polynomials

The *FFT* algorithm (Aho *et al.*, 1974; Bonneau, 1974; Moenck, 1976; Winograd, 1978) is useful for multiplying and powering dense polynomials with coefficients from the field of complex numbers or a finite computation structure (Bonneau, 1974) (typically the integers modulo a prime). The precision or size of the modulus must be decided a priori, to be at least as large as the precision of the result. This is an inconvenience in algebraic manipulation systems where the magnitude of integer coefficients can grow arbitrarily large.

The *FFT* works by computing the *discrete Fourier transform* of the univariate polynomial $p(x)$, which is the vector $[p(\omega^0), \ldots, p(\omega^{s-1})]$ where d is the degree of $p(x)$, the size of the transform $s \geq d+1$, and ω is some *principal sth root of unity.* The inverse of the transform is computed in a nearly identical way; effectively the transform vector can be regarded as the coefficients of the polynomial $p'(y) = \sum p(\omega^i)y^i$ and the vector of coefficients of $p(x)$ is $[p'(\omega^{-0})/n, \ldots, p'(\omega^{-s+1})/n]$. By evaluating two polynomials at $\omega^0, \ldots, \omega^{s-1}$, multiplying the corresponding values together, and converting the result back, the product polynomial is produced. Likewise, powering is performed by evaluating the polynomial, powering each resulting value, and converting back to get the powered polynomial.

The (Cooley-Tukey) *Fast Fourier Transform* (*FFT*) (Aho *et al.*, 1974; Bonneau, 1974) is the basic algorithm for computing the discrete Fourier transform in time $O(d \log d)$. It works as follows:

Let s be a power of 2, $s \geq d+1$

Let $A = [a_0, \ldots, a_{s-1}]$ be the coefficients of $p(x)$ (padded with zeros if necessary) in some computation structure C.

Let $A' = [a'_0, \ldots, a'_{s-1}]$ be the coefficients in the transformed polynomial $p'(y)$.

Let ω be a primitive sth root of unity in C.

$A' = FFT(A, s, \omega)$.

```
Recursive $FFT(A, s, \omega)$:
  [1] if $s = 1$, return $a_0$.
  [2] split coefficients by index into even-indexed sequence $B$ and odd-indexed
      sequence $C$.
  [3] $B' \leftarrow FFT(B, s/2, \omega^2)$, $C' \leftarrow FFT(C, s/2, \omega^2)$.
  [4] for $i \leftarrow 0$ to $s/2-1$ do
        $a'_i \leftarrow b'_i + \omega^i c'_i$.
        $a'_{i+s/2} \leftarrow b'_i + \omega^{i+s/2} c'_i$.
  [5] return $A'$.
```

Some multiplication and addition operations are hidden in the powering and the manipulation of the indices. Asymptotically, $\Theta(d \log d)$ additions and multiplications are performed since each step of the recursion works on subproblems of exactly half the size. d must be rounded up to a power of two. Numerous tricks can be used to trim the number of operations (Aho *et al.*, 1974) by a constant factor, specifically by "unravelling" the recursion and using bit-operations to permute the coefficients as necessary. An iterative form of the *FFT* can be stated as follows:

```
Iterative $FFT(A, s, \omega)$:
  [1] for $l \leftarrow 0$ to $\lg(s)$ do
  [2] for $i \leftarrow 0$ to $s-1$ do $t_i \leftarrow a_{i+1}$.
  [3] for $i \leftarrow 0$ to $s-1$ do
        $r_i \leftarrow t_{i \wedge (-2^{\lfloor \lg i \rfloor - l})} + \omega^{(s/2^{l+1}) \cdot \mathrm{bitreverse}(l)} \cdot t_{i \vee (2^{\lfloor \lg i \rfloor - l})}$.
  [4] for $l \leftarrow 0$ until $s-1$ do $a'_{\mathrm{bitreverse}(l)} \leftarrow r_l$.
  [5] return $A'$.
```

where $\lfloor \lg x \rfloor$ is the greatest integer no larger than the log base-2 of x, bitreverse(x) is the reversal of the bits in x (within the fixed word length), and $\wedge$ and $\vee$ are bitwise *and* and *or*, respectively.

The *Good-Winograd* algorithm (Bonneau, 1974; Winograd, 1978) provides another decomposition for cases where s is *not* a power of 2, but is a product of two relatively prime integers of roughly equal size. The Good-Winograd algorithm factors the degree into two relatively-prime numbers and treats the polynomial as if it were bivariate (the multidimensional algorithm is described in section (4.1)). Powers of x are implicitly replaced by power-products of the two new variables. Either the Good-Winograd or the Cooley-Tukey algorithms can be applied to the subproblems, depending upon their size. The complexity of the Good-Winograd algorithm is the same as the Cooley-Tukey algorithm.

If we consider the finite-field *FFT* in terms of bit operations, rather than integer operations (Aho *et al.*, 1974), the complexity reads somewhat higher. Letting b be the number of bits required to hold the final answer (i.e. $b \leq \lceil 2 \lg(x) \lg(s) \rceil$ for multiplication, where x is the number of bits required to hold the largest of the coefficients of the operands, or $b \leq k \lg x \lg s$ for raising the polynomial to power k), and s be a power of 2, we can operate in the ring of integers modulo $m = \omega^{s/2} + 1 \geq b$, where the principle sth root of unity ω is a power of 2. Fixing ω as $2^{8\lfloor \lg x \rfloor}$ satisfies this formula (for multiplication); then the modulus m gives us a bit field of length $b' \in O(s \lg x)$. The cost of b'-bit addition, then, is $O(b')$; b'-bit multiplication by ω (a power of 2) can also be performed in $O(b')$ bit operations. Thus the bit-complexity of the *FFT* and inverse *FFT* is $O(s^2 \lg(s) \lg(x))$.

Since the *DFT* is a *linear* transformation of coefficient vectors to "value" vectors, addition can be performed in either domain. Compound expressions of addition, multiplication, and powering operations on polynomials can be performed efficiently by transforming the initial polynomials into their corresponding *DFT*s and using pointwise addition, multiplication, and powering operations. The necessary number of evaluation points and precision must be computed ahead of time, as there is no known way to increase the size of the transform that is better than converting to coefficient form and computing the larger size transform. In fact increasing the number of evaluation points cannot be significantly easier than computing the *DFT*, since we could otherwise derive a faster *DFT* algorithm by evaluating the polynomial at one point and "filling in" points until the s-point *DFT* is formed.

Differentiation and integration of polynomials in *DFT* form are apparently most easily done by conversion back to coefficient form, performing the operation, and converting to *DFT* form again. In fact, a few inexpensive "standard" operations appear to be harder in the *DFT* form, such as identifying a zero polynomial, and computing the value of the leading coefficient. Identifying a zero polynomial takes time linear in s, since each evaluation point must be zero. Finding the sign of the leading coefficient requires finding the non-zero coefficient of highest degree, which requires looking at (and generating) all the coefficients in the worst case.

4.1. THE MULTIVARIATE *FFT*

The multivariate *DFT* of a v-variate polynomial $p(x_1, \ldots, x_v)$ is the v-dimensional vector $[p(\omega_1^{i_1}, \ldots, \omega_v^{i_v})]$, where i_k ranges from 0 to d_k. d_k is the highest degree to which the kth variable occurs in p (padded out as necessary). ω_k is a principal d_kth root of unity in the computation structure.

The multivariate *DFT* is computed by repeatedly applying the *FFT* to each variable in turn. Initially p is a v-dimensional coefficient vector $[p_{i_1 \ldots i_v}]$; in the kth iteration the partially transformed vector

$$\left[\sum_{j_1=0}^{d_1} \cdots \sum_{j_{k-1}=0}^{d_{k-1}} p_{i_1\cdots i_v}\omega_1^{i_1 j_1} \cdots \omega_{k-1}^{i_{k-1} j_{k-1}}\right]$$

is mapped into

$$\left[\sum_{j_1=0}^{d_1} \cdots \sum_{j_k=0}^{d_k} p_{i_1\cdots i_v}\omega_1^{i_1 j_1} \cdots \omega_k^{i_k j_k}\right].$$

The kth mapping amounts to $\prod_{i\neq k} d_i$ *FFT*s of size d_k+1, adding up to a total amount of work proportional to

$$\sum_{k=1}^{v} d_k \log d_k \prod_{i\neq k} d_i,$$

which for uniform $d_i = d$ reduces to $\Theta(d^v \log d^v)$.

4.2. PARALLEL IMPLEMENTATION OF THE *FFT*

Both the Cooley–Tukey and the Good–Winograd algorithms parallelize effectively. Step 3 of the recursive algorithm can be parallelized to give a parallel running time of $\Theta(d + d/k \log d/k)$ for k processors, adding up to time $\Theta(d)$ for $k \in \Theta(d)$ processors. This decomposition is suitable for message-passing multiprocessors, where each process subdivides the problem and initiates a new process for each subproblem. The more intelligent approach would be to have one process divide the coefficient array k ways in step 2 and apply the serial iterative algorithm to the subprocesses. The combining step 4 can be performed recursively in parallel as before. The asymptotic complexity remains the same but data traffic between processes is reduced, resulting in a lower constant overhead. For a fixed number of processors and "sufficiently large" input, the $d/k \log d/k$ term dominates so the speed-up is asymptotically linear in k.

On a shared-memory multiprocessor, the more efficient iterative algorithm can be parallelized in (looping) steps 2, 3, and 4, giving a running time $\Theta(\log d)$ for $\Theta(d)$ processors. The appendix gives a Lisp program implementing this version of the parallel *FFT*.

Shared memory is not absolutely necessary for an efficient parallel *FFT*. The iterative algorithm only requires certain combinations of values at each step; specific permutation networks with nearest-neighbor shared memory are sufficient. An example of such a processor is the BBN Butterfly (BBN Labs, Cambridge, Mass.).

Kung (1981) suggested using special-purpose VLSI hardware for computing the *FFT* and other functions. Such circuits (Bonneau, 1974) can be very fast, but are only good for fixed-sized input and finite computation structures with bounded modulus. Such hardware would be useful for raising the bottom level of the *FFT* recursion from size 1 to the size s handled by the *FFT* hardware, with a resulting time complexity of $\Theta(n \log n/s + n/s \log s)$ for $\Theta(s \log s)$ hardware. The speed-up is less than linear in s for increasingly large inputs. It is unlikely that special hardware would be cost-effective, and would certainly be used only with great inconvenience.

4.3. REPRESENTATIONAL ISSUES

Using linked lists of monomials to represent input polynomials (as is done in the Macsyma general representation (Fateman, 1979) would restrict parallelism the same as the lack of shared memory, since the list must be traversed in serial to separate the even- and odd-indexed coefficients on each level of the recursion.

Converting from linked list form to array form takes linear time and cannot be parallelized in any reasonable way. Conversion from array form to linked lists can be done in parallel by forming sublists from contiguous sections of the array and splicing them together.

For dense polynomials, linked list representations not only waste time and interfere with parallelism, but waste space as well. Since most systems assume sparseness by default, a conversion to and from the dense representation should be fast, and a suite of operations entirely using dense *DFT* representations and dense coefficient operations should be considered.

5. Operations on Sparse Polynomials

In this section we present algorithms for multiplying and powering sparse polynomials in parallel, which are asymptotically optimal with regard to scalar multiplication operations.

5.1. THE *SIMP* ALGORITHM FOR MULTIPLICATION

A simple way to multiply polynomials p_1 and p_2 is as follows (Fateman, 1974*a*):

```
Given polynomials $p_1, p_2$, return $p_1 \cdot p_2$.
[1] If $p_1$ is a monomial, multiply each term of $p_2$ by $p_1$ and return result. Otherwise,
[2] Split $p_1$ into $A$ and $B$, and recursively form the products $A \cdot p_2$ and $B \cdot p_2$.
[3] Merge the two partial products (ordered by exponent), adding coefficients with
    the same exponent. Return result.
```

Hence we call it the *Simp* algorithm. This effectively decomposes polynomial p_1 into size(p_1) monomials, forms the product of each monomial with p_2, and successively merges the results. The parallelized form is to perform the recursive calls in step 2 in parallel.

The number of coefficient multiplications required is size(p_1) size(p_2). The merge step amounts to a recursive balanced merge, requiring $O(\text{size}(p_1)\,\text{size}(p_2) \cdot \log(\text{size}(p_1)\,\text{size}(p_2)))$ comparisons. There are $\text{size}(p_1)\,\text{size}(p_2) - \text{size}(p_1p_2)$ additions.

In parallel, $k \le \text{size}(p_1)$ processors can be used. The recursive step 2 is better replaced by splitting p_1 k-ways and having each processor perform the multiplication as before. Step 3 reduces to performing a parallel mergesort (section 3) on the partial products. The number of parallel multiplications is $1/k\,\text{size}(p_1)\,\text{size}(p_2)$, since multiplications are performed on the bottom level of the decomposition.

The parallel mergesort uses a parallel measure of comparisons $O(\text{size}(p_1p_2) + (\text{size}(p_1p_2)/k)\log(\text{size}(p_1p_2)/k))$, which is asymptotically $O(\text{size}(p_1p_2))$ as k approaches size(p_1). Replacing the parallel merges with a (serial) k-way balanced merge gives a parallel measure of comparisons

$$O(\text{size}(p_1p_2) \cdot \log(k) + (\text{size}(p_1p_2)/k)\log(\text{size}(p_1p_2)/k)).$$

For message-passing multiprocessors with a high communication cost, this decomposition is probably superior.

Thus, given sufficient processors, the best time for the multiplications is $O(\text{size}(p_1))$ and for additions and comparisons is $O(\text{size}(p_1p_2))$. For $\text{size}(p_1) = \text{size}(p_2) = n$, this amounts to a reduction by $\Theta(n)$ in the time to perform multiplications and $\Theta(\lg(n))$ in the time to perform comparisons. In the worst case the number of additions is reduced at most by a constant factor, since $\Theta(n^2)$ terms can combine in the final merge.

5.2. THE *NOMC* ALGORITHM FOR POWERING

The *NOMC* algorithm (*full multinomial expansion with dynamic programming*) is one of many asymptotically efficient algorithms for powering sparse polynomials. Several alternatives are mentioned in section (5.3). The algorithm is expressed as follows:

```
Given a polynomial $p = (a_1 + \cdots + a_t)$ and a power $k$ to be computed, return $p^k$.
[1] if $p$ is a monomial, power it and return the result
    (e.g. $p = cx_1^{i_1} \cdots x_n^{i_n}$, where $c$ is a coefficient,
    return $p = c^k x_1^{i_1+k} \cdots x_n^{i_n+k}$).
[2] Tabulate products of powers of each $a_n$ of total degree $\lfloor k/2^i \rfloor$, $\lceil k/2^i \rceil$ for
    $i = 0, \ldots, \log(k)$ using the relation
    $a_1^{I_1} a_2^{I_2} \cdots a_t^{I_t} = (a_1^{J_1} a_2^{J_2} \cdots a_t^{J_t}) \cdot (a_1^{K_1} a_2^{K_2} \cdots a_t^{K_t})$, $I_i = J_i + K_i$,
    $I_1 + \cdots + I_t = m$, $J_1 + \cdots + J_t = \lfloor m/2 \rfloor$, $K_1 + \cdots + K_t = \lceil m/2 \rceil$.
[3] Return $\sum_{I_1+\cdots+I_t=k} \binom{k}{I_1 I_2 \cdots I_t} a_1^{I_1} a_2^{I_2} \cdots a_t^{I_t}$.
```

For k a power of 2, the number of monomials tabulated is

$$\sum_{i=1}^{\log(k)} \binom{\operatorname{size}(p) + k/2^i - 1}{\operatorname{size}(p) - 1} < \binom{\operatorname{size}(p) + k - 1}{\operatorname{size}(p) - 1} + \sum_{i=1}^{k/2} \binom{\operatorname{size}(p) + i - 1}{\operatorname{size}(p) - 1}$$
$$= \binom{\operatorname{size}(p) + k - 1}{\operatorname{size}(p) - 1} + \binom{\operatorname{size}(p) + k/2}{\operatorname{size}(p) - 1} \sim \binom{\operatorname{size}(p) + k - 1}{\operatorname{size}(p) - 1}.$$

Since each monomial is formed by multiplying two monomials from the table (using one coefficient multiply and v exponent additions), the number of coefficient multiplications is asymptotically the same as the number of monomials in the result. The higher-order term continues to dominate for k not a power of 2. Each successive multinomial coefficient is generated from a previous one using one integer multiply and up to one integer divide.

For a parallel implementation, the result monomials are broken into groups. Each processor fills in what is needed in the monomial table to compute its group of monomials. The coefficient multiplications are parallelized perfectly for up to

$$O \frac{\binom{\operatorname{size}(p) + k - 1}{\operatorname{size}(p) - 1}}{\binom{\operatorname{size}(p) + k/2}{\operatorname{size}(p) - 1}}$$

processors, since the overhead of constructing the lower-order monomials is dominated by the cost of generating the k-order monomials. For large k and increasing size(p), this bound approaches

$$\mathcal{K} = \frac{\operatorname{size}(p)^{k/2}}{k \cdot (k-1) \cdots (k/2)}$$

processors. Beyond this, the second-order term

$$\binom{\operatorname{size}(p) + k/2}{\operatorname{size}(p) - 1}$$

becomes significant and restricts the asymptotic speed-up. Therefore the best time for the multiplications is

$$\binom{\text{size}(p)+k/2}{\text{size}(p)-1}.$$

For $\mathcal{K}$ processors the parallel mergesort requires a number of comparisons asymptotically proportional to

$$\binom{\text{size}(p)+k-1}{\text{size}(p)-1}\sim\frac{\text{size}(p)^k}{k!}$$

for large k and increasing size(p). Therefore the best time for the additions and comparisons is

$$O\left(\frac{\text{size}(p)^k}{k!}\right).$$

6. Some Other Algorithms

As mentioned earlier, the *FFT* algorithm is the most efficient way known to multiply and power *dense* polynomials in serial. Several other algorithms were developed prior to the *FFT*, including *Karatsuba* (Fateman, 1974*a*; Alagar & Probst, 1987), which works by divide-and-conquer: splitting the two polynomials into equal-sized parts, and adding their partial products. Careful arrangement of additions and subtractions eliminates the need to compute one partial product for half-splitting, so $O(n^{\log_2 3})$ multiples are performed (in the dense case). Partitioning into quarters gives an algorithm requiring $O(n^{\log_4 9})$ multiples, etc. Another is the *Eval* algorithm (Fateman, 1974*a*; Knuth, 1969), a conceptual predecessor of the *FFT*. *Karatsuba* appears reasonable for sparse polynomials, in serial or in parallel, though it will degenerate to performing the same operations as *Simp* for sufficiently sparse cases.

A recent paper by Ben-Or & Tiwari (1988) describes an algorithm for *sparse* interpolation taking $O(t^2\log^2(t)+\log(nd))$ operations, where t is the number of non-zero terms in the result, n is the number of variables, and d is the maximum degree to which any variable occurs. A sparse analog to *FFT* multiplication or powering can be constructed which takes operations dependent on the number of non-zero terms. However, the *Simp* and *NOMC* algorithms are more efficient, taking a number of operations proportional to the number of terms in the result. (Furthermore, the sparse interpolation algorithm requires the use of sequences of prime numbers, which are not particularly cheap to compute).

So far we have made no distinction between *univariate* and *multivariate* polynomials. The *Simp* and *NOMC* algorithms depend on the input being uniformly sparse; a multivariate polynomial may be separated into sparse and dense components, and operated on more efficiently by a combination of algorithms favouring sparsity and density. If a polynomial non-uniformly dense in x is written as univariate in x with polynomial coefficients, a density-favouring algorithm can be used on the "backbone" of the polynomial, while a sparsity-favouring algorithm can be used to operate on the polynomial coefficients.

A recent paper by Alagar & Probst (1987) used a combination of *Simp*, *Karatsuba*, and *FFT* for multiplying multivariate polynomials. They found that *Simp* tended to outperform *Karatsuba* by about 40% for univariate cases. It is reasonable to believe that *Simp* should perform even better for sparse polynomials. Alagar & Probst's polyalgorithm should be quite efficient for multiplying polynomials of arbitrary densities, since *Simp*, *Karatsuba*, and *FFT* are each most efficient for different cases. Such a polyalgorithm should adapt well to parallelism since the *Simp*, *Karatsuba* and *FFT* can be parallelized individually. Alternately, a general library for polynomial manipulation might include a parallel subroutine for each algorithm so the user can decide based on information about the nature of the polynomial data. The presence of parallelism will probably shift the "cut-off" points which determine which algorithm will be most efficient, depending upon how well the multiplication, addition, and comparison operations parallelize with respect to each other.

For sparse powering, several algorithms are presented in Fateman (1974*b*), (1974*a*) and Probst & Alagar (1979). *NOMC* is an efficient variation of the *NOMA* and *NOMB* algorithms (Fateman (1974*b*)). The references focus on the asymptotically-efficient *BINB*, for *binomial expansion with half-splitting.* This is a divide-and-conquer algorithm partitioning the polynomial into the sum of two polynomials, and using the binomial expansion to form the result. Computing A^n by multiplying $A^{n/2} \cdot A^{n/2}$ is more expensive than by computing $A^{n-1} \cdot A$. Performing polynomial multiplications simultaneously will not balance the workload among the processors, since the binomial expansion contains power-products where the exponents are balanced all different ways. Parallelism can be used in performing each particular polynomial multiplication, but this would be applying parallelism to large numbers of small subproblems, accumulating process-spawning overhead. *NOMC* was formulated as an alternative, spawning processes at the top-level to divide the problem into equal-sized subproblems, one per processor.

7. Empirical Results

An experiment was run to compare parallelized versions of the *Simp* and *Karatsuba* algorithms, on a 4-processor Alliant running Qlisp. These were implemented as recursive algorithms with the *Simp* algorithm generating 2 processes per level of recursion and the *Karatsuba* algorithm generating 3. Each program we tested uses the parallelized algorithm up to a fixed number of processes, and the serial algorithm afterward. Table 1 shows the result in milliseconds, garbage-collection time excluded. The codes are presented in the appendices.

The *Karatsuba* algorithm achieved almost perfect linear speed-up to the number of actual processors available. The *Simp* algorithm did not quite, but still ran strictly faster as processors were added, for this input. The cut-off point for *Karatsuba* to outrun *Simp*

Table 1. Time to expand $(x^{13}+x^{12}+\cdots+x+1)^7$ with 4 processors

Number of processes	*Simp*	*Karatsuba*
1	3537	7589
2	1864	—
3	—	2375
4	1067	—
8	1100	—
9	—	1875

was between $(x^{60}+x^{59}+\cdots+x+1)^7$ and $(x^{80}+x^{79}+\cdots+x+1)^7$ (a well-coded *FFT* should outrun both for as low as $(x^5+x^4+\cdots+x+1)^3$); unfortunately the Qlisp system broke for problems that large.

A serial version of the powering problem ran in 1283 ms on a VAX 11/785 with (Franz Inc.) Common Lisp and 3537 ms on the Alliant with one-processor Qlisp; the fact that the Alliant is in other respects generally faster than the VAX suggests that Qlisp is poorly implemented.

8. Conclusions

The parallel *Simp* and *NOMC* algorithms are efficient with regard to the coefficient multiplication operations, yielding a reduction in parallel multiplications which is linear in the number of processors. Since coefficient multiplication is by far the most expensive of the scalar operations, for sparse polynomials of reasonable size we expect a nearly linear speed-up corresponding to the reduction in parallel multiplication operations. Provided "large" polynomials contain "large" coefficients, the time spent performing scalar multiplications should continue to dominate the cost of the remaining operations.

The parallel mergesorts used to combine terms are not asymptotically optimal with regard to the total number of comparisons performed, nor is the reduction in parallel comparisons linear with the number of added processors. For large numbers of processors, the number of coefficient multiplications required to multiply polynomials of size n is reduced by a factor of roughly n, while the comparisons are reduced by a factor of only $\log(n)$. For sufficiently "large" polynomials containing "small" coefficients and for large numbers of processors, the comparisons and additions begin to dominate the cost, since they are not so completely parallelized.

For a fixed number of processors and "sufficiently large" inputs, *Simp* and *NOMC* approach a linear reduction in all operations. Since parallelism is useful only for relatively large inputs—only then is the overhead of process subdivision and scheduling dominated by the parallel components of the computation—it appears that these two algorithms should be of practical value and attain a nearly linear speed-up as processors are increased in number, up to a threshold increasing in the size of the input.

The *FFT* is asymptotically the most efficient algorithm known for multiplication and powering of dense polynomials, in serial and in parallel. Its serial effectiveness has been tested in Fateman (1974*a*), Bonneau (1974) and Alagar & Probst (1987), and has been used in conjunction with other algorithms for sparse polynomials (Alagar & Probst, 1987) for a general polynomial multiplication algorithm. A simple modification of the serial approach is suitable for parallelism.

An algebraic manipulation system for solving large problems should include *FFT*-based routines for multiplying and powering large dense polynomials. Likewise, an algebraic manipulation system using multiple processors should utilize the parallel *FFT* as well.

Special thanks to Prof. Richard Fateman for supervising this research, Prof. John Canny for his comments, and Hervé J. Touati for suggestions in the analysis of *NOMC*.

This work was supported in part by the Army Research Office, grant DAAG29-85-K-0070, through the Center for Pure and Applied Mathematics, University of California, Berkeley, and the Defense Advanced Research Projects Agency (DoD) ARPA order #4871, monitored by Space & Naval Warfare Systems Command under contract N00039-84-C-0089, through the Electronics Research Laboratory, University of California, Berkeley.

References

Aho, A. V., Hopcroft, J. E., Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

Alagar, V. S., Probst, D. K. (1987). A fast, low-space algorithm for multiplying dense multivariate polynomials. *ACM TOMS* **13**(1), 35-57.

Ben-Or, M., Tiwari, P. (1988). A deterministic algorithm for sparse multivariate polynomial interpolation. *STOC 88*, 301-309.

Bonneau, R. J. (1974). *Polynomial Operations using the Fast Fourier Transform*. Ph.D. thesis, Dept. of Mathematics, Mass. Inst. of Tech., Cambridge, Mass.

Buchberger, B. (1985). Gröbner bases: an algorithmic method in polynomial ideal theory. In: *Multidimensiona Systems Theory*, (Bose, N. K., ed.) D. Reidel Publishing Co., 184-232.

Cole, R. (1986). Parallel Merge Sort. *FOCS 86*, IEEE, 511-516.

Fateman, R. J. (1974*a*). Polynomial multiplication, powers and asymptotic analysis: some comments. *SIAM J. Comput.* 3(3), 196-213.

Fateman, R. J. (1974*b*). On the computation of powers of sparse polynomials. *Studies in Applied Mathematics* **LIII**(2), 145-155.

Fateman, R. J. (1979). Macsyma's general simplifier: philosophy and operation. In: *Proc. 1979 Macsyma Users Conference* (Lewis, V. E., ed.) 563-582.

Goto, E., Kanada, Y. (1976). Hashing lemmas on time complexities with applications to formula manipulation. *SYMSAC 76*, ACM, New York, 154-158.

Harper, L. H., Payne, T. H., Savage, J. E., Strauss, E. (1975). Sorting X+Y. *CACM* **18**(6), 347-349.

Knuth, D. E. (1969). *The Art of Computer Programming, Vol. 2: Semi-Numerical Algorithms (1st ed.)*. Addison-Wesley, Reading, Mass.

Kung, H. T. (1981). Use of VLSI in algebraic computation: some suggestions. *SYMSAC 81*, 218-222.

Moenck, R. T. (1976). Practical fast polynomial multiplication. *SYMSAC 76*, 136-148.

Probst, D. K., Alagar, V. S. (1979). A family of algorithms for powering sparse polynomials. *SIAM J. oj Computing* **8**(4), 626-644.

Winograd, S. (1978). On computing the fast Fourier transform. *Math. Comput.* **32**(141), 175-199.