

## ABSTRACT

### SEQUENTIAL AND PARALLEL ALGORITHMS FOR SYMBOLIC COMPUTATION OF INTEGER POWERS OF SPARSE POLYNOMIALS

David Karl Probst

This thesis proposes four new sequential algorithms, and two new parallel algorithms, for symbolic computation of integer powers of completely or almost completely sparse polynomials in one or several variables, and analyses their time complexity and space complexity. The two parallel algorithms are specifically intended to be run on two proposed special-purpose parallel machines, also described in the thesis. We conjecture that one of the new sequential algorithms is optimal for time and space within the family of algorithms which adopt a binomial-expansion approach to computing integer powers of sparse polynomials. If the original sparse polynomial consists of  $t$  monomials, then the time complexity of computing the  $n^{\text{th}}$  power using the best new sequential algorithm is given by:

$$\frac{t^n}{n!} + t^{n-1} \left[ \frac{1}{2(n-2)!} + \frac{1}{2^{n-2}(n-1)!} \right] + o(t^{n-2}), \text{ for } n > 2$$

The space complexity for the same task and the same algorithm is given by:

$$\frac{t^{n-1}}{2^{2n-4}(n-1)!} + O(t^{n-2}), \text{ for } n > 2$$

The two new parallel algorithms, one for a multiprocessor machine and one for an associative-processor machine, both have speed-up ratios which asymptotically approach the theoretical ideal of a speed-up ratio of  $N$  for  $N$  processors. The aim here has been, not to exhibit optimal algorithms, but merely to demonstrate the extreme suitability of a parallel approach to this problem. The space complexities of the parallel algorithms do not greatly exceed the space complexities of their sequential counterparts. We give arguments which suggest that the associative-processor architecture may be the preferred architecture of the two.

5,

ACKNOWLEDGEMENTS

## ACKNOWLEDGEMENTS

I gratefully acknowledge the invaluable assistance of my thesis adviser, Prof. V.S. Alagar, in all phases of the preparation of this thesis. In particular, I acknowledge Dr. Alagar's suggestion of the problem, and constant collaboration thereafter. This collaboration took the form of the two of us proposing ideas, and constantly discussing and re-discussing them. Nearly all of the results which have been obtained here are the fruit of these discussions. I also acknowledge considerable financial assistance from Dr. Alagar.

In addition, I should like to acknowledge the contribution of the Department Chairman, Prof. H.S. Heaps, in two respects: (1) initial financial support, and (2) Prof. Heaps' policy of actively encouraging research by students in the Department.



TABLE OF CONTENTS

7  
TABLE OF CONTENTS

	PAGE
ABSTRACT . . . . .	i
ACKNOWLEDGEMENTS . . . . .	iii
LIST OF FIGURES . . . . .	vi
LIST OF TABLES . . . . .	vii
NOTATIONS . . . . .	viii
 CHAPTER I INTRODUCTION . . . . .	 1
CHAPTER II NOTATION AND ELEMENTARY RESULTS . . . . .	11
CHAPTER III STATE OF THE PROBLEM PRIOR TO THE PRESENT WORK . . . . .	19
3.1 Algorithm RMUL . . . . .	20
3.2 Algorithm RSQ . . . . .	20
3.3 Algorithm NOMA . . . . .	22
3.4 Algorithm NOMB . . . . .	22
3.5 Algorithm BINA . . . . .	23
3.6 Algorithm BINB . . . . .	24
 CHAPTER IV THE ALGORITHM FAMILY . . . . .	 26
4.1 Design Decisions . . . . .	29
4.1.1 General Approach . . . . .	29
4.1.2 Splitting . . . . .	30
4.1.3 Powers of Subpolynomials . . . . .	34
4.1.4 Combining Powers . . . . .	37
4.1.5 Optimum Design Decisions . . . . .	39
 CHAPTER V TIME COMPLEXITY OF ALGORITHMS (SEQUENTIAL OPTION) . . . . .	 41
5.1 Algorithm BINB . . . . .	41
5.2 Algorithm BINC . . . . .	43
5.3 Algorithm BINS . . . . .	48
5.4 Algorithm BIND . . . . .	49
5.5 Algorithm BINE . . . . .	54
5.6 Algorithm BINF . . . . .	66

CHAPTER VI	SPACE COMPLEXITY OF ALGORITHMS (SEQUENTIAL OPTION)	73
6.1	Space Analysis for BINA and BINB	78
6.2	Space Analysis for BINE and BINF	82
CHAPTER VII	PARALLEL ALGORITHMS	95
CHAPTER VIII	MULTIPROCESSOR OPTION (MULTIPROCESSOR E)	100
8.1	Description of the Architecture	100
8.2	Description of the Algorithm	105
8.3	Analysis of the Speed-Up Ratio	108
CHAPTER IX	ASSOCIATIVE-PROCESSOR OPTION (ASSOCIATIVE F)	114
9.1	Description of the Architecture	114
9.2	Description of the Algorithm	120
9.3	Analysis of the Speed-Up Ratio	125
CHAPTER X	CONCLUSION	133
REFERENCES		137
APPENDIX I	VALUES OF $B(t,n)$ , $C(t,n)$ , $D(t,n)$ , $E(t,n)$ , AND $L(t,n)$ FOR SELECTED VALUES OF $t$ AND $n$	139
APPENDIX II	VALUES OF $E(t,n) - F(t,n)$ FOR SELECTED VALUES OF $t$ AND $n$	141
APPENDIX III	VALUES OF $S_B(t,n)$ , $S_E(t,n)$ , $S_{\bar{E}}(t,n)$ , AND $S_F(t,n)$ FOR SELECTED VALUES OF $t$ AND $n$	142

LIST OF FIGURES

## LIST OF FIGURES

NUMBER		PAGE
4.1	Three term-group trees . . . . .	35
4.2	The program <sup>2</sup> family tree . . . . .	39
5.1	The power group triangle when $n = 6$ . . .	57
5.2	A term-group tree . . . . .	59
5.3	A term-group tree . . . . .	62
8.1	System diagram . . . . .	101
8.2	Power group triangle (pgt) for $n = 4$ . .	104
9.1	System diagram . . . . .	115
9.2	Pgt for $n = 5$ . . . . .	117

LIST OF TABLES

## LIST OF TABLES

NUMBER		PAGE
2.1	Values of a few Stirling numbers of the first kind . . . . .	13
5.1	Itemizing the costs in BINB . . . . .	42
5.2	Itemizing the costs in BINC . . . . .	44
5.3	Leading terms for $B(t,n)$ , $C(t,n)$ , $L(t,n)$ .	47
5.4	Itemizing the costs in BIND . . . . .	50

NOTATIONS



## NOTATIONS

f	Original polynomial
g	Arbitrary polynomial
k	Logarithm of t (base 2) <u>or</u> dummy variable
n	The power sought
p	$\lfloor n/2 \rfloor$ <u>or</u> dummy variable
r	$\lfloor n/2 \rfloor$ <u>or</u> dummy variable
s	Subnode size <u>or</u> dummy variable
t	Number of monomials in f
a-list, b-list	In $\binom{n}{r} f_1^r f_2^{n-r}$ , if $\text{size}(f_1^r) > \text{size}(f_2^{n-r})$ , then $f_1^r$ is the a-list, and $\binom{n}{r} f_2^{n-r}$ is the b-list. Ties are broken by the form of the pg
group(s,n)	$\sum_{j=1}^n \text{size}(s,j)$
pgt	Power group triangle
size(f)	Number of monomials in the polynomial f
size(s,n)	An alternate functional form. If $\text{size}(f) = s$ , this is synonymous with $\text{size}(f^n)$
t <sub>1</sub> fields, t <sub>2</sub> fields	Registers in the PPA
AM	Associative memory
BB	Binomial-binomial work
BC(s,n)	Binomial coefficient work to process a node whose subnodes are of size s
BINA, ..., BINF	Sequential algorithm names
BW	Binomial work

$B(t,n), \dots$ $F(t,n)$	Sequential algorithm time complexities
CU	Control unit
$E$	Number of CM words for exponent set
GBW	Group, binomial work
$L(t,n)$	Theoretical lower limit for sequential algorithm time complexity
$M$	$N/2^j$ (Multiprocessor architecture only)
MIMD	Multiple instruction stream, multiple data stream
$M_n$	Equals 1(3) when $n$ is even(odd)
Multiprocessor E Associative F	Parallel algorithm names
$N$	Parallel processor multiplicity. (Multi-processor and associative-processor architectures)
NBW	Nonbinomial work
$N_n$	Equals 1(0) when $n$ is even(odd)
N-split	Dividing an amount of work among $N$ processors. There are multiprocessor $N$ -splits and PPA $N$ -splits,
$P$	Number of CM words for link field
PPA	Parallel processing array
$P(t,n),$ $T(t,n)$	Parallel algorithm time complexities
RBW	Root binomial work
$S_A, \dots$ $S_F$	Sequential algorithm space complexities
SIMD	Single instruction stream, multiple data stream

$\delta_p$	Equals 0(1) when $p$ is even (odd)
$\sigma_j^{(n)}$	The $j^{\text{th}}$ symmetric function over $n$
$\binom{n}{r}$	Binomial coefficient $n$ choose $r$
$[n_r]$	Stirling number of the first kind
$\{n_r\}$	Stirling number of the second kind
$O(\dots)$	Big $O$ notation of Bachmann
$\lfloor \dots \rfloor$	Floor of ...
$\lceil \dots \rceil$	Ceiling of ...

To my wife and our  
forthcoming child

CHAPTER I  
INTRODUCTION

## CHAPTER I

### INTRODUCTION

This thesis contains results arising from the investigation and analysis of new algorithms for computing powers of symbolic polynomials on both sequential and parallel machines; it is intended as a contribution to theoretical computational complexity which would not be irrelevant to practical problems of algorithm assessment and programming. Previous attempts to determine best algorithms for polynomial powering on sequential machines have been motivated by a desire to obtain system programs for use in symbolic algebraic manipulation systems. Parallel polynomial powering algorithms have been much less studied, if at all; the aim here is to devise more or less special-purpose computer architectures, and then formulate parallel algorithms especially suited to them. It is noteworthy that efficient serial algorithms are not necessarily extendable to obtain efficient parallel algorithms. The parallel algorithms presented here are obtained by finding or creating parallelism in the sequential algorithms, and then exploiting it.

The choice of an appropriate computer algorithm will determine the size and complexity of problems which can be solved in a reasonable time. The approach taken in this thesis is to choose one out of a set of competing algorithms on the basis of a theoretical analysis of the intrinsic algorithm

complexity rather than, say, on the basis of run-time tests. In order that the problems of algorithm analysis be well-posed, it is necessary to specify both a computational model, which characterizes the problem domain, and a cost model, which provides the criterion or criteria by which intrinsic difficulty is to be measured. The computational model is that the input polynomials be completely sparse polynomials (definition follows) in one or more variables (indeterminates) with integer coefficients. The cost model is that the algorithms will be analysed in terms of the number of coefficient multiplications required to compute the final result - in the parallel case, in terms of the number of parallel cycles needed to perform these multiplications. Both models will now be explained.

Completely dense and completely sparse polynomials are the two basic complementary computational models for which one does an analysis of powering algorithms. Sparse polynomials have few non-zero coefficients; completely dense polynomials have no zero coefficients. A univariate (one-variable) polynomial of degree  $d$  is completely dense if it has no zero coefficients. That is, it has  $d + 1$  terms. A multivariate polynomial with  $v$  variables where each variable  $X_1, \dots, X_v$  occurs to maximum degree  $d$  will be completely dense if all possible terms are present. In that case,

$$f = f_d X_v^d + \dots + f_0$$

where the  $f_i$  are completely dense in  $v-1$  variables, and

$$\text{size}(f) = (d+1)^v,$$

where  $\text{size}(f)$  is the number of monomial terms in  $f$ . If  $f^i$  has  $(d+1)^v$  terms for  $1 \leq i \leq n$ , then  $f^i$  remains completely dense to power  $n$ . This is the worst case assumption of polynomial growth. Dense polynomials in 3 or more variables can only be raised to quite small powers before either core or time become excessive. [10]

In sparse polynomials, whatever the number of variables (indeterminates) in the polynomial, and whatever the degree of the polynomial in each variable (indeterminate), there are only  $t$  non-zero terms. We represent a polynomial in the class of sparse polynomials as a sum of monomials, where each monomial is of the form  $c(\prod x_i^{\alpha_i})$ ,  $\alpha_i \geq 0$ ,  $c$  and  $\alpha_i$  integers. A sparse polynomial may be characterized by the number of non-zero terms,  $t$ , in its representation as a sum of monomials. We say that a polynomial  $f$  is completely sparse to power  $n$  if  $f^i$ , fully expanded, for all  $i$ ,  $1 \leq i \leq n$ , contains exactly the number of terms of the  $t$ -term multinomial expansion. To say that the number of terms is given by the size of the  $t$ -term multinomial expansion is to say that no further collection of like terms is possible. Multivariate problems dealt with by symbolic algebraic manipulation systems are often sparse in character. The theoretical model of 'sparse polynomial' is due to Morven Gentleman. [7]



4

The assumption in the computational model of complete sparsity of the input polynomials affects the design and analysis of the powering algorithms in two ways. From the standpoint of design, sparsity guarantees that no like terms will ever be formed through multiplying one subpolynomial by another, given the particular structure of the new binomial-expansion algorithms. This will be explained below. From the standpoint of analysis, sparsity allows us to predict the sizes of all intermediate results, and thus carry through the analysis in a mathematically tractable way. If the input polynomial is more or less sparse, but not completely sparse, then the analysis remains correct, but the algorithms, as they now stand, produce results in which the relatively few like terms have not been collected.

We have said that the computing time will be analysed for completely sparse polynomials with integer coefficients; actually, it is only necessary that the coefficients be 'non-growing', to allow us to assume that the cost of coefficient arithmetic is constant. One way to guarantee this is to take the coefficients from a finite field. In this way, multiplication and addition costs do not grow with the number of digits in the result. Alternatively, one may suppose that all coefficient arithmetic is single-precision floating-point or single-precision integer arithmetic. To fix the model, we choose integer arithmetic.

The proposed cost model implies that the algorithm be analysed in terms of the number of coefficient multiplications required to compute the final result. This model will now be justified. The cost of an algorithm may be taken to be the number of elementary operations which occur during the computation. When the overhead is minimal, this reduces to the number of elementary arithmetic operations. The cost of multiplying two sparse polynomials is the cost of multiplying each term of one by each term of the other, and then collecting any like terms which may have been formed. Since the sparsity of the polynomials implies that the number of like terms is small, the cost of addition in actually collecting the like terms is negligible. This is for the general case. For the new algorithms presented in this thesis, wholly based on the principle of binomial expansion, no like terms are formed through the multiplication of sparse polynomials. Hence, there are no like terms to be collected.

To see this last point, consider computing  $f^n$  as

$$f_1^n + f_2^n + \sum_{r=1}^{n-1} \binom{n}{r} f_1^r f_2^{n-r},$$

where  $f_1$  and  $f_2$  are the two halves of  $f$ . Imagine that the powers of both  $f_1$  and  $f_2$  have been computed. It can be shown that if any of the terms in the products of the summation combine, then  $f$  is not completely sparse to power  $n$ , contradicting the hypothesis. This will be proven below. In all of the algorithms presented in this

thesis, the fundamental arithmetic work consists, either of multiplying a binomial coefficient by a subpolynomial, or of multiplying a subpolynomial by another subpolynomial and then not collecting like terms, which do not exist in the completely sparse case. The total computation, therefore, consists of some number of monomial multiplications (coefficient multiplication plus exponent set addition) and some number of binomial coefficient multiplications (coefficient multiplication without exponent set addition.) The sum of these two numbers is the number of coefficient multiplications required by the algorithm, and also the number we use to measure the cost of the algorithm.

There is empirical evidence, as well as theoretical arguments, in favour of the cost model adopted in this thesis. Richard Fateman [5] has made theoretical and experimental analyses of algorithms for multiplication and powering of dense symbolic polynomials. These algorithms involved multiplications, divisions, additions, subtractions, and exponentiations. Comparing results of timings with actual counts of the above operations showed that actual computation times were closely proportional to the total of these counts and reasonably proportional to just the multiplications/divisions. In the binomial-expansion algorithms developed in this thesis, the only elementary operations are (1) product of monomial times monomial, and (2) product of binomial coefficient times monomial. For each of the algorithms presented, an exact count is given of the number of such products which occur.

This thesis is intended as a contribution to theoretical computational complexity. It may not be inappropriate, however, to stress again the practical concerns which underlie much work in symbolic algebraic manipulation. Symbol manipulation systems, such as the MACSYMA system at Project MAC or the Altran symbolic manipulator at Bell Telephone Laboratories, were developed as practical, cost-effective systems for actual computation. The designers of such systems were interested in the analysis of algorithms because they wanted sufficient understanding of the relative merits of competing algorithms to make concrete, practical decisions, viz., which algorithms to implement as system programs most suitable to the actual computations to be undertaken by their system. In this context, therefore, the design and analysis of algorithms has roots in practical problems existing, in some sense, outside of computer science.

Several algorithms for computing integer powers of sparse polynomials on sequential machines have been given recently by Fateman [4], and represent the state-of-the-art prior to the current work. The principal aims of this thesis are severalfold: (1) to exhibit superior algorithms for computation on sequential machines, (2) to provide insight, based on analysis and leading to intuition, as to why certain algorithms are better than others, (3) to make an exhaustive complexity analysis, both with respect to time and with respect to space, (4) to devise special-purpose computer architectures for parallel computation, and (5) to modify and

adapt the sequential algorithms to obtain parallel algorithms especially suited to the special-purpose computer architectures. The best parallel algorithms are then compared with the best sequential algorithms, to obtain the speed-up ratio, or the ratio of computation times on sequential and parallel machines.

Each of the algorithms presented in this thesis, whether for computation on a sequential machine or for computation on a parallel machine (two examples are provided in this thesis), takes a binomial-expansion approach to the symbolic computation of integer powers of sparse polynomials, as this approach is judged optimum. An exhaustive analysis of the whole family of binomial-expansion algorithms is undertaken. In this problem area, it is difficult, if not impossible, to prove that any one algorithm is optimal, even given a complete specification of the computational model, the cost model, and the general approach to be taken. However, in all of the binomial-expansion algorithms that we have been able to imagine, there is one, or perhaps two, which are vastly superior to all the rest. It is unlikely that the leading terms of their cost functions can be bettered. In the parallel case, we have two parallel algorithms, to be run on two special-purpose parallel architectures, both of whose speed-up ratios approach the theoretical upper limit in an asymptotic sense. In an even stronger asymptotic sense, the complexity of the sequential algorithms approaches the theoretical lower limit.

The family of sequential binomial-expansion algorithms will be explored systematically. According to Dijkstra [3]:

'A program should be conceived and understood as a member of a family ....'

Compare Parnas [13]:

'The aim of the new design methods is to allow the decisions which can be shared by a whole family, to be made before those decisions, which differentiate family members.'

Trees are convenient representations of program family structures. The root is the problem to be solved, and the terminal nodes are the fully-specified algorithms for doing so. The branches which descend from a node are the alternative design decisions which may be taken at that point. Such trees allow great insight into the relative costs of algorithms. In the parallel case, parallel binomial-expansion algorithms are developed for multiprocessor and associative processor architectures. The parallel algorithms and architectures are compared with each other, and with their sequential counterparts. In reality, we favour the associative processor architecture, even though its *prima facie* speed-up ratio is less. It was not possible to implement the parallel algorithms because the envisaged parallel machines have not been built yet.

In what follows, after collecting certain analytical results, there will be (1) a statement of the problem before the current work was undertaken, i.e., a survey of previous

results, (2) an exploration of the family of sequential binomial expansion algorithms, i.e., a systematic enumeration of design alternatives, (3) descriptions of the sequential algorithms, (4) analysis of how to minimize time in the sequential case, (5) analysis of how to minimize space in the sequential case, (6) discussion of the criteria by which the success and efficiency of parallel algorithms and computer systems are judged, (7) descriptions of special-purpose multi-processor and associative-processor architectures, (8) descriptions of the parallel algorithms, (9) analysis of the speed-up ratio and space complexity of the parallel algorithms, and finally, (10) some comments about actual implementations of the most successful sequential binomial-expansion algorithm. In all that follows, a balance will be sought between analytic detail and informative general statements which provide a broad perspective.

A number of the new results appearing in this thesis have already been published in [2]. In large measure, the first five chapters of the thesis are an elaboration of sections from the earlier paper. In particular, most of the new results in Chapters IV and V appear already in [2]. The exceptions are the closed form for  $E(t,n)$ , and the entire discussion of BINP, unknown at the time the earlier paper was submitted. The new results in Chapters VI, VIII and IX have not yet been submitted for publication, and appear here for the first time.

## CHAPTER II

### NOTATION AND ELEMENTARY RESULTS



## CHAPTER II

## NOTATION AND ELEMENTARY RESULTS

In this section we will state a number of elementary mathematical results, theorems, notations, and closed form expressions which will be of considerable use in subsequent analysis. The following standard theorems are quite useful.

$$\text{Theorem 2.1} \quad \sum_{i=0}^n \binom{r+i}{r} = \binom{r+n+1}{r+1} = \binom{r+n+1}{n}$$

$$\text{Theorem 2.2} \quad \sum_{i=0}^n \binom{r+i}{r} \binom{r+n-i}{r} = \binom{2r+n+1}{n}$$

**Theorem 2.3** Let  $f$  be a  $t$ -term polynomial which is completely sparse to power  $n$ . If  $\text{size}(f)$  gives the number of non-zero terms in  $f$ , then

$$\text{size}(f^n) = \binom{t+n-1}{n}, \quad t \geq 2$$

The proof is given, e.g., in Fateman [4].

$$\text{Theorem 2.4} \quad \sum_{i=1}^{n-1} \binom{r+i}{r} \binom{r+n-i-1}{r-1} = \binom{2r+n}{2r} - \binom{r+n-1}{r} - \binom{r+n}{r}$$

The proof is not difficult.

**Theorem 2.5** The multiplication of a polynomial  $f$  by a polynomial  $g$  can be done with a cost of  $\text{size}(f) \cdot \text{size}(g)$ .

Proof:

If  $f$  and  $g$  are two sparse polynomials, then the cost of obtaining their product is the cost of multiplying each term of one by each term of the other, and then collecting the like terms in the product so formed. (The assumption here, for completely sparse polynomials, is that straightforward classical multiplication is difficult to improve upon.) Inasmuch as the sparsity of polynomials implies that the number of like terms is small, the cost of addition in actually collecting the like terms is negligible. The cost, therefore, may be taken as  $\text{size}(f) \cdot \text{size}(g)$ , if the cost of the merge sort is ignored. More rigorously, if it is known a priori that there are no like terms, then the cost will be  $\text{size}(f) \cdot \text{size}(g)$ .

Consider the fact that

$$\sum_{i=1}^{n-1} \binom{t/2+i-1}{t/2-1} \binom{t/2+n-i-1}{t/2-1} = \binom{t+n-1}{n} - 2 \binom{t/2+n-1}{n}$$

This implies, in

$$f^n = (f_1 + f_2)^n = f_1^n + f_2^n + \sum_{i=1}^{n-1} \binom{n}{i} f_1^i f_2^{n-i},$$

where  $\text{size}(f_1) = \text{size}(f_2) = t/2$ , that no terms in any product may combine, because, if they did,  $f^n$  would no longer have the proper size for a completely sparse polynomial.

Let us review the notation for Stirling numbers of the first kind. Our reference here is Knuth [12].

TABLE 2.1 VALUES OF A FEW STIRLING NUMBERS OF THE FIRST KIND

	$\begin{bmatrix} n \\ 1 \end{bmatrix}$	$\begin{bmatrix} n \\ 2 \end{bmatrix}$	$\begin{bmatrix} n \\ 3 \end{bmatrix}$	$\begin{bmatrix} n \\ 4 \end{bmatrix}$	$\begin{bmatrix} n \\ 5 \end{bmatrix}$
$n = 1$	1	0	0	0	0
$n = 2$	1	1	0	0	0
$n = 3$	2	3	1	0	0
$n = 4$	6	11	6	1	0
$n = 5$	24	50	35	10	1

The following are important results concerning these numbers.

$$\begin{bmatrix} n \\ 1 \end{bmatrix} = (n-1)! \quad \begin{bmatrix} n \\ n \end{bmatrix} = 1 \quad \begin{bmatrix} n \\ n-1 \end{bmatrix} = \begin{bmatrix} n \\ 2 \end{bmatrix} = \frac{1}{2} n(n-1)$$

$$\begin{bmatrix} n \\ m \end{bmatrix} = (n-1) \begin{bmatrix} n-1 \\ m \end{bmatrix} + \begin{bmatrix} n-1 \\ m-1 \end{bmatrix}, \quad n > 0$$

E.g.,

$$s(s+1)(s+2) = 2s + 3s^2 + s^3$$

$$= \sum_{j=1}^3 \begin{bmatrix} 3 \\ j \end{bmatrix} s^j$$

In general,

$$s(s+1)(s+2) \dots (s+p) = \sum_{j=1}^{p+1} \begin{bmatrix} p+1 \\ j \end{bmatrix} s^j \quad (2.6)$$

Let us now state and prove some preliminary closed forms of useful expressions which arise later in the analysis of these algorithms.

The function  $\text{size}(f)$  has already been introduced. Let us change the notation slightly so that  $\text{size}(s, n)$  is  $\text{size}(f^n)$  when  $\text{size}(f) = s$ . As usual, sparsity is assumed. We have

$$\text{size}(s, n) = \binom{s+n-1}{n} = \frac{1}{n!} \sum_{j=1}^n \begin{bmatrix} n \\ j \end{bmatrix} s^j \quad (2.7)$$

Proof:

$$\begin{aligned} \binom{s+n-1}{n} &= \frac{(s+n-1)!}{n!(s-1)!} \\ &= \frac{1}{n!} (s+n-1) \dots (s) \\ &= \frac{1}{n!} \sum_{j=1}^n \begin{bmatrix} n \\ j \end{bmatrix} s^j \quad \text{by (2.6)} \end{aligned}$$

We need a notation for the size of a collection or group of powers. Let  $\text{group}(s, n)$  be

$$\sum_{j=1}^n \text{size}(s, j)$$

i.e., the sum of sizes of all powers from 1 to  $n$  of a polynomial whose size is  $s$ . We have

$$\text{group } (s, n) = \binom{s+n}{n} - 1 = \frac{1}{n!} \sum_{j=1}^n \left[ \begin{matrix} n+1 \\ j+1 \end{matrix} \right] s^j \quad (2.8)$$

Proof:

$$\begin{aligned} \sum_{j=1}^n \binom{s+j-1}{j} &= \binom{s+n}{n} - 1 \\ &= \frac{(s+n)!}{n! s!} - 1 \\ &= \frac{1}{n!} \cdot \frac{1}{s} \cdot (s+n) \dots (s) - 1 \\ &= \frac{1}{n!} \cdot \frac{1}{s} \cdot \sum_{j=1}^{n+1} \left[ \begin{matrix} n+1 \\ j \end{matrix} \right] s^j - 1 \quad \text{by (2.6)} \\ &= \frac{1}{n!} \sum_{j=1}^n \left[ \begin{matrix} n+1 \\ j+1 \end{matrix} \right] s^j, \end{aligned}$$

since

$$\left[ \begin{matrix} n+1 \\ 1 \end{matrix} \right] = n!,$$

where  $j$  has been replaced by  $j+1$ . The coefficient of the leading term, viz.,  $\left[ \begin{matrix} n+1 \\ n+1 \end{matrix} \right]$ , is 1.

Another useful closed-form expression is the following.

$$\binom{s+n}{n-1} - n = \frac{1}{(n-1)!} \sum_{j=1}^n \sum_{k=0}^{n-j-1} \left[ \begin{matrix} n-1 \\ k+j \end{matrix} \right] \binom{k+j}{k} 2^k s^j \quad (2.9)$$

where, as a convention,  $\sum_{k=0}^{-1} \dots$  is zero.

Proof:

$$\begin{aligned}
 \binom{s+n}{n-1} - n &= \frac{(s+n)!}{(n-1)!(s+1)!} - n \\
 &= \frac{1}{(n-1)!} (s+n) \dots (s+2) - n \\
 &= \frac{1}{(n-1)!} \sum_{j=1}^{n-1} \binom{n-1}{j} (s+2)^j - n \quad \text{by (2.6)} \\
 &= \frac{1}{(n-1)!} \sum_{j=1}^{n-1} \binom{n-1}{j} \sum_{k=0}^j \binom{j}{k} s^k 2^{j-k} - n \\
 &= \frac{1}{(n-1)!} \sum_{j=1}^{n-1} \binom{n-1}{j} \sum_{k=1}^j \binom{j}{k} s^k 2^{j-k},
 \end{aligned}$$

since the constant term in  $(s+n) \dots (s+2)$  is  $n!$ .

Expansion and collection of like terms gives finally

$$\binom{s+n}{n-1} - n = \frac{1}{(n-1)!} \sum_{j=1}^n \sum_{k=0}^{n-j-1} \binom{n-1}{k+j} \binom{k+j}{k} 2^k s^j,$$

where the term for  $j=n$  contributes nothing, as explained above.

The closed-form expressions which have just been introduced are instrumental in obtaining a closed-form expression for the time complexity of the best sequential binomial-expansion algorithm. The function  $\text{size}(s,n)$  is used continually in sparse polynomial time complexity analysis; together with the function  $\text{group}(s,n)$  it also gives an important tool for sparse polynomial space complexity analysis.

Finally, there is a theorem, suggested by Michael Fredman [4], which gives a lower bound on the number of multiplications required to compute a power of a polynomial. That number is at least equal to  $\text{size}(f^n) - \text{size}(f)$ .

**Theorem 2.10** No algorithm can compute  $f^n$ , the  $n^{\text{th}}$  power of an arbitrary polynomial, in fewer than  $\text{size}(f^n) - \text{size}(f)$  multiplications.

The proof may be found in Fateman [4]. Suppose now that  $f$  is completely sparse, and that  $\text{size}(f) = t$ . According to Theorem 2.10, the lower limit on the number of multiplications required by any algorithm to compute  $f^n$ , which will be denoted by  $L(t, n)$ , is given by

$$\begin{aligned} L(t, n) &= \text{size}(t, n) - t = \binom{t+n-1}{n} - t \\ &= \frac{1}{n!} \sum_{j=1}^n \binom{n}{j} t^j - t \quad \text{by (2.7)} \end{aligned}$$

This formula may also be written as

$$L(t, n) = \frac{t^n}{n!} + \frac{t^{n-1}}{2(n-2)!} + O(t^{n-2})$$

This is the least number of multiplications possible. When one has a good algorithm, one compares the number of multiplications it uses with the theoretical lower limit to see how far off one is. The cost functions for BINE and BINF,

these latter being the two best sequential binomial-expansion algorithms presented in this thesis, express the number of multiplications used by these algorithms. The leading terms of either cost function are

$$\frac{t^n}{n!} + t^{n-1} \left[ \frac{1}{2(n-2)!} + \frac{1}{2^{n-2}(n-1)!} \right] + o(t^{n-2}),$$

for  $n > 2$ . We will show by comparison with the cost functions of other algorithms just how good this is.



### CHAPTER III

#### STATE OF THE PROBLEM PRIOR TO THE PRESENT WORK

## CHAPTER III

STATE OF THE PROBLEM PRIOR TO THE  
PRESENT WORK

The best previously-existing sequential algorithm for computing powers of sparse polynomials is due to Fateman [4]. It is one of several algorithms for this problem he analysed to obtain insight into their relative merits. We shall reproduce his descriptions of these algorithms and adopt his analyses, except in the case of his best algorithm, which will be analysed anew. Another computationally much less interesting, i.e., much more expensive, algorithm for computing powers of symbolic polynomials, due to Horowitz and Sahni, will also be mentioned. Fateman has considered repeated multiplication (RMUL), repeated squaring (RSQ), two specific approaches based on multinomial expansion (NOMA and NOMB), and two specific approaches based on binomial expansion (BINA and BINB), all as methods for computing integer powers of sparse polynomials. The main conclusion is, that of these six algorithms, algorithm BINB is computationally most efficient. In general, the analysis suggests that binomial expansion is the most promising general approach. This analysis led to a research programme to develop those superior binomial-expansion algorithms for serial and parallel computation which are presented in this thesis.

### 3.1 ALGORITHM RMUL (REPEATED MULTIPLICATION)

#### Description:

RMUL successively computes  $f^2 = f.f$ ,  $f^3 = f.f^2, \dots$ ,  $f^n = f.f^{n-1}$ . Cf. Gentleman [7].

#### Analysis:

By Theorem 2.5, the cost for the polynomial multiplications is

$$\begin{aligned} \text{size}(f) \cdot \sum_{i=1}^{n-1} \text{size}(f^i) &= t \cdot \sum_{i=1}^{n-1} \binom{t+i-1}{t-1} \\ &= t \cdot \binom{t+n-1}{t} - t \end{aligned} \quad (3.1.1)$$

by Theorems 2.3 and 2.1.

### 3.2 ALGORITHM RSQ (REPEATED SQUARING)

#### Description:

When  $n$  is a power of 2, RSQ computes  $f^n$  as  $f^2 = f.f$ ,  $f^4 = f^2.f^2, \dots$ ,  $f^n = f^{n/2}.f^{n/2}$ . When  $n$  is not a power of 2, it computes a sequence of powers of  $f$  based on the expansion of  $n$  as a binary number. More formally

- (1)  $q \leftarrow 1$ .  $z \leftarrow f$ .
- (2)  $i \leftarrow$  rightmost bit of  $n$ . Shift  $n$  right one bit.
- If  $i = 0$ , go to (4).

- (3) If  $q = 1$ ,  $q \leftarrow z$  else  $q \leftarrow q.z$ .
- (4) If  $n = 0$ , return  $q$  else  $z \leftarrow z.z$  and go to (2).

That this algorithm correctly computes  $f^n$  by means of binary expansion of the power  $n$  may be seen by considering  $n$  as a binary number and observing the relationship between multiplication of powers and addition of exponents.

#### Analysis:

The analysis is done for  $n$ , a power of 2. Even in this case, RMUL is superior to RSQ. When  $n$  is not a power of 2, this superiority (of RMUL) merely increases. When  $n = 2^j$ , the cost for RSQ is

$$\sum_{i=0}^{j-1} \text{size}(f^{2^i})^2 = \sum_{i=0}^{j-1} \binom{t+2^i-1}{t-1}^2 \quad (3.2.1)$$

For  $n = 4$ , RSQ costs  $\frac{1}{4}(t^4 + 2t^3 + 5t^2)$  whereas RMUL costs  $\frac{1}{6}(t^4 + 6t^3 + 11t^2)$ . For  $t > 7$ , RMUL is cheaper. For  $n = 8$ , RMUL is better for  $t > 3$ . Both Fateman [4] and Gentleman [7] discuss the nature of the superiority of RMUL to RSQ. Fateman remarks that it is less costly to multiply a large polynomial by a small polynomial, than to multiply two polynomials of intermediate size.

### 3.3 ALGORITHM NOMA' (FULL MULTINOMIAL EXPANSION - A)

#### Description:

Let  $f = a_1 + \dots + a_t$  where the  $a_i$ ,  $1 \leq i \leq t$ , are monomials. Assume that we have precomputed the  $n^{\text{th}}$  power of  $g = \alpha_1 + \dots + \alpha_t$  where the  $\alpha_i$  are new symbols (not in  $f$ ). Then all we need to do is to substitute  $a_i$  for  $\alpha_i$  in  $g^n$ . We can compute the substitutions using Horner's rule. For the detailed algorithm description, refer to Fateman's paper [4].

#### Analysis:

The cost which is obtained is

$$\sum_{j=0}^n t \binom{t+j-2}{t-1} = t \binom{t+n-1}{t} \quad (3.3.1)$$

### 3.4 ALGORITHM NOMB' (FULL MULTINOMIAL EXPANSION - B)

#### Description:

For each of the  $t$  monomials in  $f$ ,  $a_1, \dots, a_t$ , compute a list  $\ell_i = \langle a_i^0, a_i^1, \dots, a_i^n \rangle$ . Next, compute all products consisting of combinations of no more than one element  $a_i^{n_i}$  from each list such that  $n_1 + \dots + n_t = n$ . This product is then multiplied by the multinomial coefficient

$$\binom{n}{n_1, \dots, n_t}$$

For more details, see Fateman [4].

Analysis:

The cost which is given is

$$t \binom{t+n-2}{t-1} + tn - 2t \quad (3.4.1)$$

3.5 ALGORITHM BINA (BINOMIAL EXPANSION WITH MONOMIAL SPLITTING)

Description:

The polynomial  $f$  is split into two parts  $f_1 + f_2$ .

The size of  $f_1$  is 1, and the size of  $f_2$  is  $t-1$ . To compute  $f^n$ , we compute  $f_1^2, \dots, f_1^n$  and  $f_2^2, \dots, f_2^n$ , and use the binomial theorem

$$f^n = \sum_{i=0}^n \binom{n}{i} f_1^i f_2^{n-i}$$

Note that  $f_2^2$  can be calculated by BINA, and other powers are simply computed by  $f_2^i = f_2 \cdot f_2^{i-1}$ . See Fateman [4].

Analysis:

Itemizing the steps and their costs, and adding gives finally the cost for BINA,  $n \geq 2$ . It is

$$t \binom{t+n-2}{t-1} - \frac{1}{2}(t^2 - 3t + 10) + 2n \quad (3.5.1)$$

### 3.6 ALGORITHM BINB (BINOMIAL EXPANSION WITH HALF SPLITTING)

#### Description:

BINB is identical in concept to BINA, but  $f$  is split as evenly as possible into  $f_1 + f_2$ . See Fateman [4].

#### Analysis:

When Fateman itemizes the steps and their costs, and adds, he obtains as the cost for BINB

$$\binom{t+n-1}{n} + t \binom{t/2+n-1}{n-1} - 2 \binom{t/2+n-1}{n} - t/2(t/4-n-1) - \log_2(t-1)+4 \quad (3.6.2)$$

We repeat the entire analysis below and obtain a slightly higher cost function for BINB.

RMUL is clearly cheaper than RSQ. Comparing (3.1.1) with (3.3.1) we see that, surprisingly, NOMB is no better than RMUL. Comparing (3.4.1) with (3.1.1), we see that NOMB/RMUL is approximately  $t/(t+n-1)$ . That is, for a given  $t$ , NOMB gets arbitrarily better as  $n$  increases. Yet NOMB is difficult to program. Comparing (3.5.1) with (3.1.1) and (3.4.1) gives: BINA/NOMB is approximately 1, and BINA/RMUL is approximately  $t/(t+n-1)$ . Again, for fixed  $t$  and increasing  $n$ , BINA becomes arbitrarily better than RMUL. Finally,

comparing (3.6.2) with (3.5.2), and calculating the leading terms in both cases, we see that BINB is consistently better than BINA. The leading term of the latter is  $t^n/(n-1)!$ ; the leading term of the former, according to Fateman [4], is

$$t^n \left[ \frac{1-2^{1-n}}{n!} + \frac{2^{1-n}}{(n-1)!} \right].$$

These cost functions demonstrate clearly, that among these six algorithms, algorithm BINB is the most computationally efficient. Questions of efficiency and ease of programming make the binomial-expansion algorithms appear as good choices for computing powers of sparse polynomials. Fateman conjectures that an algorithm superior to BINB may be hard to find. The conjecture is, of course, false. Yet BINB is not, by any means, a bad algorithm insofar as it does approach the theoretical lower limit for coefficient multiplications when  $t$  and  $n$  become large. In this context, the multinomial-expansion algorithm for powering sparse polynomials proposed by Horowitz and Sahni[10] is considerably less interesting. Their cost function is only of the order of the theoretical lower limit, in one case being roughly equal to four times that limit. It is possible to interpret this result as even further evidence for the superiority of the binomial-expansion approach.



CHAPTER IV  
THE ALGORITHM FAMILY

## CHAPTER IV

### THE ALGORITHM FAMILY

Professor Dijkstra has been quoted above to the effect that:

"A program should be conceived and understood as a member of a family ..."

The reasoning here, relevant both to algorithm design and to algorithm analysis, is that choices, the design decisions taken in any algorithm, can be evaluated only if one sees them against the background of a range of possible alternatives. One evaluates a specific design decision which is taken in comparison with other design decisions which could have been taken. Ultimately, one evaluates a specific algorithm which is proposed in comparison with other algorithms which could have been proposed. In this way, one comes to understand the structure of a family of algorithms which solve the same problem. A tree structure, branching downwards from the root, is an extremely convenient graphical representation of the algorithm family structure, of its decision points and possible decisions. The root of the tree represents the problem to be solved. The terminal nodes represent fully-specified algorithms, some better, some worse, which solve the problem. In general, each non-terminal node of the tree represents a point at which a decision must be taken, a choice situation, while the branches which descend downwards from that node represent the possible decisions at that point, in that situation.

At the root of the program family tree may be placed a description or specification of the problem to be solved. In our case it is, construct an algorithm which accepts an input power  $n$ ,  $n$  a positive integer, and an input polynomial  $f$ ,  $f$  completely or almost completely sparse to power  $n$ , and produces as its output the resultant polynomial  $f^n$ . As mentioned previously, only a sub-family, namely the family of sequential binomial-expansion algorithms, will be systematically explored using the program family tree. Binomial-expansion algorithms are considered the most promising; parallel algorithms will be considered later in the thesis. The chief analytic aim associated with the program family tree is to make statements about the relative costs of the various design decisions associated with different branching points over and above the analytically-obtained cost functions which allow us to choose among the fully-specified algorithms on the basis of their time complexity. A knowledge of these relative costs allows us to explain the superiority (least time-cost) of one particular algorithm as having resulted from consistently lowest-cost decisions. It also adds weight to the conjecture that this particular algorithm, the best so far, is optimal. This optimality is guaranteed if no superior design decision exists at any branching point; it will be destroyed if and when someone succeeds in imagining a design decision superior to those considered here.

For a while, attention will be restricted to, and arguments concern, the sequential case. When attention is redirected to the parallel case, the parallel designer must be prepared to revise any decision which, although clearly correct in the sequential case, is likely to hold back parallelism in the parallel case. There is a body of analysis, confirmed by results obtained here, which suggests that binomial-expansion is superior to other general approaches used in symbolic powering of sparse polynomials. This analysis has been done for the sequential case; no one has analytic results which suggest that binomial expansion is also the most promising parallel approach. The parallel algorithms developed in this thesis are binomial-expansion algorithms, the chief reason for this being primarily the extreme attractiveness of the sequential binomial-expansion algorithms. Although the speed-up ratios of the parallel algorithms both approach the theoretical upper limit, this is no guarantee of optimality of approach in the parallel case. With respect to the design decisions within the binomial-expansion sub-family, it has generally been found that those decisions which minimize computing time in the sequential case do not interfere with the possibilities of exploiting parallelism in the parallel case. The very good parallel algorithms, then, make use of the ideas which have been developed for the best sequential algorithms.

## 4.1 DESIGN DECISIONS

In this section, we explain the four major design decision areas which arise when one considers algorithms for computing integer powers of sparse polynomials. The program family tree is a graphical accompaniment which records the design decisions which differentiate algorithms of the family, and so serves to relate these algorithms to each other.

### 4.1.1 General Approach

Every algorithm proposed in this thesis takes a binomial expansion approach to computing integer powers of sparse polynomials. This means that if one has an input polynomial  $f$ , which is to be raised to some power  $n$ , one splits  $f$  into two subpolynomials  $f_1$  and  $f_2$  (which together, contain all the monomials of the original polynomial), obtains the appropriate powers of the subpolynomials, and then combines the subpolynomial powers together with binomial coefficients according to the theorem of binomial expansion to obtain the final answer,  $f^n$ . That is,  $f^n$  is computed as

$$f^n = (f_1 + f_2)^n = \sum_{r=0}^n \binom{n}{r} f_1^r f_2^{n-r}$$

Binomial expansion has been chosen in preference to some other, general approach, such as repeated multiplication, repeated squaring, or some form of multinomial expansion, not because every binomial-expansion algorithm is superior to every non-

binomial-expansion algorithm, but rather because the best binomial-expansion algorithms are superior to the best algorithms in each of the other general approaches. The position here is that binomial expansion allows a lower cost refinement in comparison with the refinements available in other approaches. As a graphical convention in connection with the program family tree, we agree to draw the branch which represents the least cost, or possibility of least cost, on the left. The first two branches of the tree, then, which descend from the root, are 'binomial expansion' on the left, and 'some other approach' on the right. The binomial-expansion family is extremely rich in algorithms. BINA and BINB, the two previously-published binomial-expansion algorithms, are only two of many possible refinements. These refinements differ considerably in cost. It is the position here that BINB, the best previously-published binomial-expansion algorithm, does not take optimal design decisions by a wide margin. The obvious strategy to improve on BINB, indeed, to look for the optimal algorithm, is to consider a broad range of possible refinements of this most attractive general approach.

#### 4.1.2 Splitting

In binomial expansion the original polynomial  $f$  is split into two parts,  $f_1$  and  $f_2$ . The relative sizes of  $f_1$  and  $f_2$  affect the time complexity of a binomial-expansion algorithm in two distinct and unrelated ways: on the one hand

they affect the time complexity of generating powers of subpolynomials, and on the other hand they affect the time complexity of combining appropriate generated powers, suitably multiplied by binomial coefficients, in the binomial expansion. These effects are so large that an algorithm with optimal splitting and a poor technique for generating powers may outperform another algorithm with suboptimal splitting and a good technique for generating powers. Some of the ways of splitting polynomials into two parts are: (a) one term and the rest, (b) as evenly as possible, (c) as unevenly as possible, a power of two and the rest, when the number of terms to be split is not a power of two, else evenly, and finally, (d) as evenly as possible, a power of two and the rest. Considering many ways of splitting is consistent with the idea of investigating a broad range of possible refinements. We found that degree of evenness of splitting is the only relevant splitting parameter in choosing from among the four splittings mentioned.

The need to choose the relative sizes of the subpolynomials  $f_1$  and  $f_2$ , where  $f = f_1 + f_2$ , follows from the very idea of binomial expansion. There is another splitting decision, however, which arises only if one considers alternative ways of generating powers of subpolynomials. The two binomial-expansion algorithms BINA and BINB adopt repeated multiplication as their principal strategy for computing powers of subpolynomials. It is characteristic of this approach that the original polynomial  $f$  is split only once; we call this

characteristic: 'single-level splitting'. Single-level splitting is a consequence of a 'non-recursive' approach to the problem, in which binomial expansion is used to compute the power of the whole polynomial, and something entirely different is used to compute the powers of subpolynomials. If, on the other hand, one carries through and uses binomial expansion to compute the powers of subpolynomials, one needs to split the original polynomial over and over again, until finally the monomial level is reached. This is called: 'multilevel splitting'. All the algorithms presented in this thesis use variants of binomial expansion for generating powers of subpolynomials, and hence are committed to multilevel splitting, in contrast to previously-published binomial-expansion algorithms, which all adopt the single-level splitting approach.

Analysis of the fully-specified algorithms allows one to make two assertions about the preferred design decisions relating to splitting. It is less costly to multiply a large polynomial by a small polynomial, than to multiply two polynomials of intermediate size. It follows that even splitting increases the cost of combining computed powers of subpolynomials. All the techniques for computing subpolynomial powers, whether variants of binomial expansion or simply repeated multiplication, are such that even splitting reduces the cost of computing these powers. In all instances, the latter effect predominates. Hence, in either a single-level or a multilevel context, polynomial splitting which is as even as



possible is always to be preferred. This is the first assertion. From the standpoint of time complexity, repeated multiplication is a poor way to compute powers of subpolynomials. Both the variants of binomial expansion, to be discussed shortly, easily outperform repeated multiplication for this task. Hence, in comparison with single-level splitting, multilevel splitting is to be preferred. And this is the second assertion.

Of two sequential algorithms which are equally parallelizable, that is, possess identical speed-up ratios, the least-cost sequential algorithm will give rise to the least-cost parallel algorithm. There is no evidence which suggests that even splitting reduces the possibilities for parallelism. On the contrary, even splitting breaks problems into subproblems of equal sizes, which can then be processed simultaneously. One may also say that even splitting is fully consistent with trying to maximize the number of independent subtasks, an important step in extracting parallelism from a sequential algorithm. Repeated multiplication, and hence single-level splitting, has the following disadvantage in the parallel case. Because, even for completely sparse polynomials, there is no theorem which asserts that no like terms are formed when generating subpolynomial powers through repeated multiplication, this latter technique is less easily parallelizable (because of the need to collect like terms) than the multilevel binomial-expansion techniques, for which it can be shown, as discussed previously, that no like terms are formed in the

cross products of the binomial expansion. Thus far at least, the same decisions seem reasonable in both the sequential and parallel cases; no further claim is made.

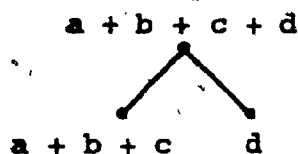
#### 4.1.3 Powers of Subpolynomials

Powers of subpolynomials must first be computed before they can be combined in the binomial expansion together with binomial coefficients. What strategy best computes these subpolynomial powers is one of the least immediately apparent issues in the whole binomial-expansion family, and yet one central to the time complexity. Previous binomial-expansion algorithms have essentially used repeated multiplication for this task; the new algorithms presented and analysed in this thesis use the techniques of recursion and binary merge. These techniques will now be explained. As soon as the decisions concerning depth (level) and evenness of splitting have been taken, the splitting of the original polynomial is fully specified, and may be displayed as a binary tree. A 'term group' is one or more terms from the original polynomial; the tree is called the 'term group tree'. The whole polynomial is placed at the root. New nodes (subnodes) are formed by taking the term group at a node, and splitting it into halves of appropriate sizes (determined by the evenness decision) to form the left and right subnodes. Convention: let the size of the term group in the left subnode never be smaller than that in the right subnode. When multilevel splitting has been adopted, this process is repeated until one reaches the individual terms,

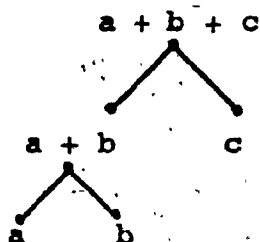
which are the terminal nodes of the term group tree.

Examples:

Single-level, one-and-the-rest splitting



Multilevel, as-even-as-possible splitting



Multilevel, even splitting with  $t = 2^k$

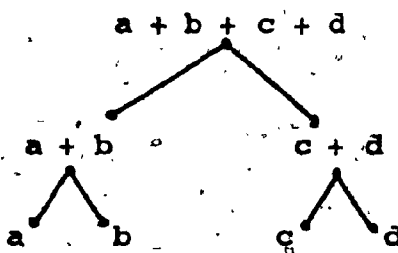


FIG. 4.1 THREE TERM-GROUP TREES

Recursion properly so-called and binary merge are the two 'recursive' approaches to the computation of powers of subpolynomials. They are consistent applications of binomial expansion to the computation of powers of polynomials at all stages, except, of course, at the monomial level itself. The idea of recursion is to obtain the powers of the subpolynomials by recursive application of the total algorithm to the subnodes. The weakness of this approach, still vastly superior to repeated multiplication, is that one never computes single powers of subpolynomials, but rather groups of all powers from two to  $n$ . Separate recursive application for distinct powers of a subpolynomial leads to recomputation: distinct powers of a binomial have computable factors in common. Binary merge is a form of recursion in which there is no recomputation. One may compute the power of a node, or the group of powers (from two to  $n$ ) of a node, if one has the groups of powers (from two to  $n$ ) of both subnodes. Equivalently, once one has computed groups of powers of subnodes, one may go on to compute groups of powers (or single powers) of the father node. In binary merge we progress up the tree, starting from the terminal nodes, using the already computed powers of the subnodes (once they have been computed) to compute the groups of powers of the father nodes. Ultimately, a single power of the root node, which contains the original polynomial, is computed.

#### 4.1.4 Combining Powers

We assume that a table of binomial coefficients  $\binom{r}{s}$ ,  $0 \leq s \leq r \leq n$ , is available to the computation. References to this table are overhead. According to the cost model, each multiplication of a monomial by a binomial coefficient counts as one coefficient multiplication. If the powers of the two subpolynomials  $f_1$  and  $f_2$ , where  $f = f_1 + f_2$ , are available,

$$f^n = (f_1 + f_2)^n = \sum \binom{n}{r} f_1^r f_2^{n-r}$$

may be obtained by combining suitable powers with binomial coefficients in the binomial expansion. The product  $\binom{n}{r} f_1^r f_2^{n-r}$  may be obtained either (a) left to right, irrespective of the relative sizes of  $f_1^r$  and  $f_2^{n-r}$ , or (b) by first multiplying  $\binom{n}{r}$  by the smaller of the two polynomials, and then this result by the remaining polynomial. Previous algorithms have invariably used the first technique; our best algorithms use the second technique. It is immediate that the second technique reduces the number of coefficient multiplications. We call the first technique: 'left-to-right', and the second technique: 'smaller'. If the first technique is used, the product may also be written as  $f_2^{n-r} \cdot \binom{n}{r} f_1^r$ . That is, one always multiplies by the power of  $f_1$  first. In the second technique, one chooses the order of multiplication for each product.

If one were to list the preferred choices which have been made so far, one would have, first, the use of binomial expansion, second, splitting the polynomial as evenly as possible, third, adopting one of the 'recursive' approaches, which commits one to multilevel splitting, fourth, preferring binary merge to recursion properly so-called, and last, forming the cross products using the smaller technique. These choices specify an algorithm, which we have called BINE, about which it is conjectured that the leading terms of the time-complexity cost function cannot be bettered. There is, however, one final improvement which can be made. Suppose that  $a$  and  $b$  are the left and right subnodes of the root of the term group tree, and that  $b_1$  and  $b_2$  are the left and right subnodes of  $b$ . We will use the same names for a node, and for the term group associated with that node. Suppose that we are computing the fourth power of the root. In that case, we are interested in the product  $a^2 \cdot 6b^2$ . It is not necessary to have previously computed  $b^2$  as  $b_1^2 + b_2^2 + b_1 \cdot 2b_2$ . Rather, one may compute  $6b^2$  directly as  $6b_1^2 + 6b_2^2 + b_1 \cdot 12b_2$ . One saves as many coefficient multiplications as there are terms in the product  $b_1 \cdot b_2$  at the cost of multiplying 6 by 2. This technique is called: 'distribution'; it amounts to not having to compute some of the lower powers of the left and right subnodes of the root. Details of the technique will be given below in Section 5.6, which analyses the time savings due to distribution. Distribution is a mixed strategy which deviates from pure binomial expansion at the highest levels of the term group tree. If the

algorithm which differs from BINE only in that distribution is used, be called BINF, then BINF is the least-cost sequential binomial-expansion algorithm (for computing powers of sparse polynomials) known at the time of writing.

#### 4.1.5 Optimum Design Decisions

If we think of distribution as a modified form of merge, then the whole set of optimum design decisions can be diagrammed in the following way:

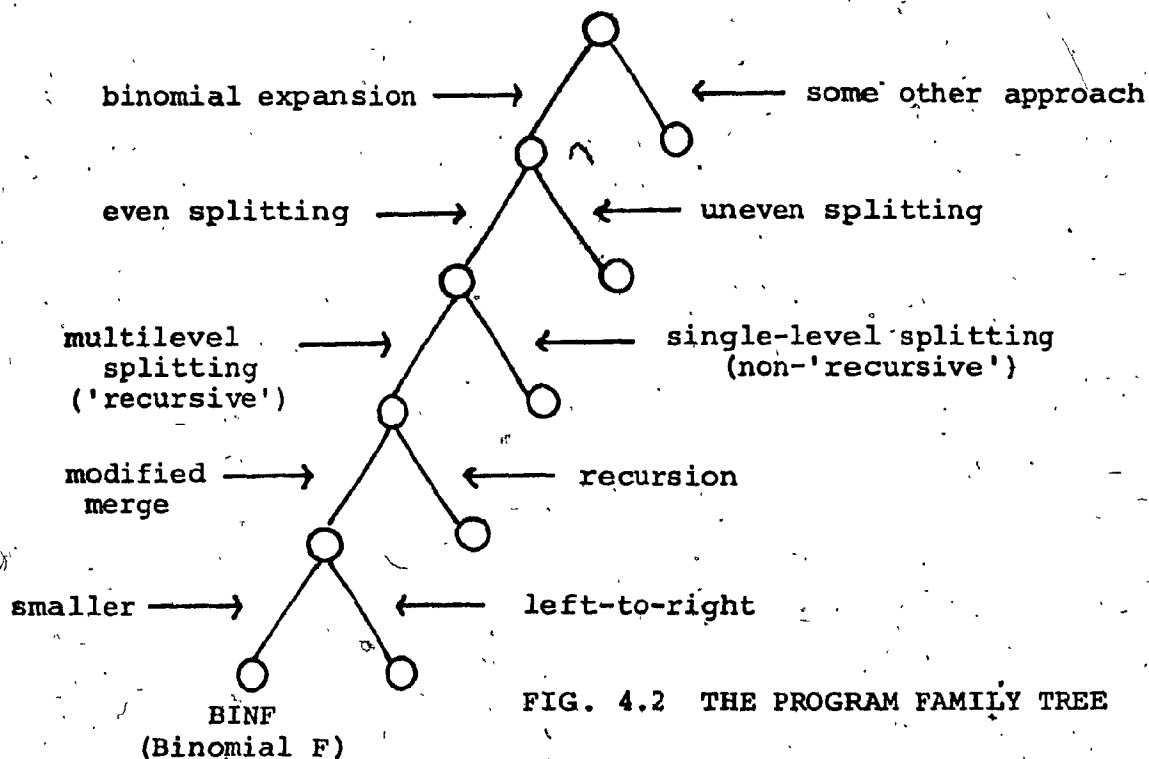


FIG. 4.2 THE PROGRAM FAMILY TREE

The diagram which has been given is a very partial and incomplete representation, even of the sub-family of binomial-expansion algorithms. The reader may imagine the combinatoric possibilities in permuting choices. Two algorithms will differ in cost if even one decision is not the same. In the

next section we present and analyse some representative algorithms, most of them new. The superiority of BINF will be demonstrated; if no leftmost branch above can be improved upon, then this algorithm is indeed optimal.

Before passing to the section for formal description and analysis of algorithms, we may use the program family tree for an initial comparison between previously existing binomial-expansion algorithms and those presented in this thesis. Obviously, all belong to the binomial-expansion subfamily. Fateman's BINA employs monomial splitting (the most uneven splitting), while his BINB splits as evenly as possible. Most of the new algorithms use even splitting; those that do not pay a time penalty. BINA and BINB are both non-recursive, single-level-splitting algorithms, as repeated multiplication generates the powers of subpolynomials. All the new algorithms are 'recursive', multilevel-splitting algorithms, using, variously, merge, modified merge, or recursion for the generation of subpolynomial powers. BINA and BINB use left-to-right. Many of the new algorithms use smaller; those that do not pay a time penalty. The program family tree is an important complement to the exact, analytically-obtained cost functions. It gives us a capsule view of the full range of differences between any two algorithms. It also provides a graphical representation of which decisions are to be preferred at each decision point. In this way, we come to understand why certain algorithms outperform others.



CHAPTER V

TIME COMPLEXITY OF ALGORITHMS  
(SEQUENTIAL OPTION)

## CHAPTER V

TIME COMPLEXITY OF ALGORITHMS  
(SEQUENTIAL OPTION)

We discuss below Fateman's BINB, which we analyse anew, and four algorithms which are all refinements of the binomial-expansion approach. Each of the four algorithms is superior to BINB. The algorithms will be described using the terminology developed in connection with the program family tree of Chapter IV. For each algorithm, the input is a power  $n$ , and a polynomial  $f$  completely sparse to power  $n$ , while the output is the resultant polynomial  $f^n$ . The aim of the time-complexity analysis is to give the cost in coefficient multiplications as a function of  $t$  and  $n$  for each algorithm.

5.1 ALGORITHM BINB (Binomial B)

This algorithm is specified by the design decisions: binomial expansion, even splitting, single-level splitting, insofar as recursion is used for the squares of subpolynomials and repeated multiplication for all other powers, and finally, left-to-right. The cost function proposed in [4] is

$$B(t, n) = \binom{t+n-1}{n} + t \binom{t/2+n-1}{n-1} - 2 \binom{t/2+n-1}{n} - \\ - t/2 (t/4 - n - \log_2(t-1) + 4) \quad (5.1.1)$$

As stated, BINB uses a mixed strategy for computing powers of

subpolynomials: recursion for the squares, and repeated multiplication for higher powers. There may be no deep philosophy behind this; a remark by Fateman in another paper [5] shows he did not think it made much difference. After closely inspecting the steps in [4, p.152-153], we would itemize the total cost of this algorithm as follows:

Case 1:  $t = 2^k$   $f = f_1 + f_2$   $\text{size}(f_1) = \text{size}(f_2) = t/2$

TABLE 5.1 ITEMIZING THE COSTS IN BINB

Step	Compute	Cost
1	$f_1^2, f_2^2$	$t^2/4 + kt/2$
2	$f_1^3, \dots, f_1^n$ $f_2^3, \dots, f_2^n$	$2 \cdot t/2 \sum_{i=2}^{n-1} \binom{t/2+i-1}{t/2-1}$
3	$\binom{n}{1} f_2$	$t/2$
4	$\binom{n}{2} f_2^2, \dots, \binom{n}{n-1} f_2^{n-1}$	$\sum_{i=2}^{n-1} \binom{t/2+i-1}{t/2-1}$
5	$f_1 \binom{n}{i} f_2^{n-i}$ , $1 \leq i \leq n-1$	$\sum_{i=1}^{n-1} \binom{t/2+i-1}{t/2-1} \binom{t/2+n-i-1}{t/2-1}$

Total Cost

$$B_1(t, n) = \binom{t+n-1}{n} - 2 \binom{t/2+n-1}{n} + (t+1) \binom{t/2+n-1}{n-1}$$

$$- t/2(t/2 - \log_2 t + 1) - 1 \quad (5.1.2)$$

Case 2:  $t \neq 2^k$   $f = f_1 + f_2$   $\text{size}(f_1) = \lceil t/2 \rceil$   
 $\text{size}(f_2) = \lfloor t/2 \rfloor$

Itemizing the cost as before we get

$$B_2(t, n) = \binom{t+n-1}{n} - \binom{t_1+n-1}{n} - \binom{t_2+n-1}{n} + \\
+ t_1 \binom{t_1+n-1}{n-1} + (t_2+1) \binom{t_2+n-1}{n-1} + p(t) \quad (5.1.3)$$

where  $p(t)$  is a polynomial in  $t$  of degree 2.

Comparing (5.1.1) with (5.1.2) we note that, even for  $t = 2^k$ , the cost we calculate is greater than the reported cost. This difference will show up in the second leading term of the cost function  $B(t, n)$ .

## 5.2 ALGORITHM BINC (Binomial C)

This algorithm is specified by the design decisions: binomial expansion, even splitting, multilevel splitting, as recursion is used to generate all powers of all subpolynomials other than monomials, and finally, left-to-right. The analysis will be carried through for  $t = 2^k$ . The original polynomial  $f$  is split into two parts,  $f_1$  and  $f_2$ , where

$$\text{size}(f_1) = \text{size}(f_2) = t/2$$

We use the binomial theorem  $f^n = \sum_{r=0}^n \binom{n}{r} f_1^r f_2^{n-r}$  to compute  $f^n$ .

The powers of the subpolynomials (other than monomials) are obtained by recursive application of the algorithm just specified.

### 5.2.1 Analysis of BINC

Let  $C(t, n)$  be the cost in coefficient multiplications of computing  $f^n$  when  $\text{size}(f) = t$ . (The cost function is always the last letter of the algorithm name.) If we expand  $f^n$  in the form

$$f^n = f_1^n + f_2^n + \sum_{r=1}^{n-1} \binom{n}{r} f_1^r f_2^{n-r}$$

we have

$$C(t, n) = 2C(t/2, n) + \sum_{r=1}^{n-1} Q_r \quad (5.2.1)$$

where  $Q_r$  is the sum of costs itemized as follows:

TABLE 5.2 ITEMIZING THE COSTS IN BINC

Step	Compute	Cost
1	$f_1^r$	$C(t/2, r)$
2	$f_2^{n-r}$	$C(t/2, n-r)$
3	$\binom{n}{r} f_1^r$	$\binom{t/2+r-1}{t/2-1}$
4	$\binom{n}{r} f_1^r f_2^{n-r}$	$\binom{t/2+r-1}{t/2-1} \binom{t/2+n-r-1}{t/2-1}$

$$C(t,n) = 2C(t/2,n) + 2 \sum_{r=1}^{n-1} C(t/2,r) + \sum_{r=1}^{n-1} \binom{t/2+r-1}{t/2-1} + \sum_{r=1}^{n-1} \binom{t/2+r-1}{t/2-1} \binom{t/2+n-r-1}{t/2-1} \quad (5.2.2)$$

Using (2.1) and (2.2) we obtain

$$C(t,n) = \binom{t+n-1}{n} - 2 \binom{t/2+n-1}{n} + \binom{t/2+n-1}{n-1} - 1 + 2 \sum_{r=1}^n C(t/2,r) \quad (5.2.3)$$

We may bootstrap a closed form for  $C(t,n)$  into one for  $C(t,n+1)$  by using the formula:

$$C(t,n+1) = nt + \frac{t-1}{n+1} \binom{t+n-1}{n} + \sum_{r=1}^k 2^{r-1} \binom{t/2^r+n-1}{n} + \sum_{r=1}^k 2^{r-1} C(t/2^{r-1},n) \quad (5.2.4)$$

obtained from (5.2.3) by changing  $n$  to  $n+1$  and using recurrence on  $t$ . Using this formula and induction on  $n$ , we obtain the general form of  $C(t,n)$ .

$$C(t,n) = \sum_{i=1}^n a_i^{(n)} t^{n-i+1} + t \sum_{i=n+1}^{2n-1} a_i^{(n)} k^{i-n} \quad (5.2.5)$$

a polynomial in  $t$  and  $k$ . Inspection yields

$$a_1^{(n)} = 1/n! \quad a_{2n-1}^{(n)} = 1/2(n-1)!$$

Exact analysis yields the following for  $n = 2, 3$ , and  $4$ :

$$C(t, 2) = t^2/2 + t/2 + kt/2 \quad (5.2.6)$$

$$C(t, 3) = t^3/6 + 5t^2/4 + 7t/12 + t(k+k^2/4) \quad (5.2.7)$$

$$C(t, 4) = t^4/24 + t^3/3 + 65t^2/24 - t/12 + t(31k/24 + 5k^2/8 + k^3/12) \quad (5.2.8)$$

For an analysis of the two leading terms of the cost function  $C(t, n)$ , we need to know the general form of the second coefficient  $a_2^{(n)}$ . This is the coefficient corresponding to the coefficient of  $t^n$  in  $C(t, n+1)$ .

Using (2.7) we see that the coefficient of  $t^n$  in

$$\frac{t-1}{n+1} \binom{t+n-1}{n} \text{ is } \frac{n(n-1) - 1}{(n+1)!}$$

Similarly, after some manipulation, we obtain the coefficient of  $t^n$  in

$$\sum_{l=1}^k 2^{r-1} \binom{t/2^r + n-1}{n} \text{ as } \frac{1}{2n! (2^{n-1} - 1)}$$

Substituting (5.2.5) into (5.2.4) and rearranging yields the coefficient of  $t^n$  in

$$\sum_{l=1}^k 2^{r-1} C(t/2^{r-1}, n) \text{ as } \frac{2^{n-1}}{n! (2^{n-1} - 1)}$$

Finally, adding and changing  $n+1$  to  $n$ , i.e.,  $a_2^{(n+1)}$  to  $a_2^{(n)}$ , we obtain the coefficient of the second leading term. It is

$$a_2^{(n)} = \frac{1}{2(n-2)!} + \frac{3}{(n-1)! (2^{n-1} - 2)}$$

The coefficient of the leading term, from  $\frac{t-1}{n+1} \binom{t+n-1}{n}$ , is  $a_1^{(n)} = 1/n!$ . Compare the analysis in [2]. We can now compare the leading terms of the cost functions  $B(t,n)$  and  $C(t,n)$  with the lower-limit cost function  $L(t,n)$ . We have:

TABLE 5.3 LEADING TERMS FOR  $B(t,n)$ ,  $C(t,n)$ ,  $L(t,n)$

$n$	$B(t,n)$	$C(t,n)$	$L(t,n)$
2	$5t^2/8$	$t^2/2$	$t^2/2$
3	$t^3/4$	$t^3/6$	$t^3/6$
$n > 3$	$[\frac{1}{n!} + \frac{2^{1-n}}{n(n-2)!}]t^n$	$t^n/n!$	$t^n/n!$

It is instructive to calculate the relative deviations of  $B(t,n)$  and  $C(t,n)$  from the minimum cost  $L(t,n)$ . We have:

$$[C(t,n) - L(t,n)]/L(t,n) = 3n/(2^{n-1}t) + O(t^{-2})$$

and

$$[B(t,n) - L(t,n)]/L(t,n) = (n-1)t/2^{n-1} + O(1)$$

For large  $t$  and large  $n$ ,  $C(t,n)$  approaches  $L(t,n)$  much more rapidly. In fact, for  $n > 2$  and  $t \geq 16$ , BINB outperforms BINB. (See Appendix I). The fact that there are (at least) four algorithms which are successively better and better than



BINB points up the weakness in the argument that, because  $B(t,n)$  is asymptotically  $L(t,n)$ , therefore BINB may be difficult to improve upon. Asymptotic arguments can be misused. The important thing is to obtain analytically-exact cost functions, and to make a coefficient by coefficient comparison of their leading terms.

### 5.3 ALGORITHM BINS (Binomial S)

We considered an algorithm which is identical to BINC, except that balanced binary splitting was used in place of even splitting. That is, whenever the size of the polynomial to be split was not a power of 2, the polynomial was split as evenly as possible into two polynomials such that the size of one of them was a power of 2. This splitting is less even than even splitting. Hence, it represents a regression with respect to BINC and is, in fact, more expensive. This less attractive algorithm still outperforms BINB for  $t > 12$ . We merely quote the final cost function. Let us suppose that the balanced binary multilevel splitting is  $t = t_1 + \dots + t_p$ ,  $t_i = 2^{k_i}$ . Let  $t^{(0)} = t$ ,  $t^{(i)} = t^{(i-1)} - t_i$ ,  $i = 1, \dots, p-1$ . In this notation  $S(t,n)$  is

$$S(t,n) = \sum_{r=0}^{p-2} S(t^{(r)}, n-1) + \sum_{r=1}^p C(t_r, n) + \sum_{r=1}^{p-1} T_r \quad (5.3.1)$$

where

$$T_r = 2 \binom{t_r+n-2}{n-1} + \binom{t^{(r-1)}+n-2}{n} - \binom{t^{(r)}+n-2}{n} - \binom{t_r+n+1}{n}$$

See [2] for more detail.

#### 5.4 ALGORITHM BIND (Binomial D)

This algorithm is specified by the design decisions: binomial expansion, even splitting, multilevel splitting, recursion, and smaller. That is, it is BINC in which left-to-right has been replaced by smaller. More formally:

- (1) Split  $f = f_1 + f_2$ ,  $\text{size}(f_1) = \lceil t/2 \rceil = t_1$   
 $\text{size}(f_2) = \lfloor t/2 \rfloor = t_2$
- (2) For  $r = 2$  to  $n$ , compute  $f_1^r$  and  $f_2^r$  using BIND unless the  $f_i$  is monomial.
- (3) For  $r = 1$  to  $n - 1$ 
  - (a) Multiply  $\binom{n}{r}$  by whichever of  $f_1^r$  and  $f_2^{n-r}$  has fewer terms
  - (b) Multiply this product by the remaining factor.
- (4) Collect the terms of the binomial expansion.

#### Analysis of BIND:

As before,  $D(t, n)$  is the cost in coefficient multiplications of obtaining  $f^n$  when  $\text{size}(f) = t$ . The costs are

itemized as follows:

TABLE 5.4 ITEMIZING THE COSTS IN BIND

Step	Cost
2	$\sum_{r=1}^n [D(t_1, r) + D(t_2, r)]$
3(a)	$\sum_{r=1}^{n-1} \min \left[ \binom{t_1+r-1}{r}, \binom{t_2+n-r-1}{n-r} \right]$
3(b)	$\sum_{r=1}^{n-1} \binom{t_1+r-1}{r} \binom{t_2+n-r-1}{n-r}$

$D(t, n)$  is the sum of expressions in the right-hand column.

We restrict the analysis to  $t = 2^k$ . However,  $D(t, n)$  has been tabulated for  $t \neq 2^k$ . (See Appendix I).

$$\begin{aligned}
 D(t, n) = & 2 \sum_{r=1}^n D(t/2, r) + 2 \sum_{r=1}^m \binom{t/2+r-1}{r-1} + \delta_n \binom{t/2+n/2-1}{n/2} + \\
 & + \sum_{r=1}^{n-1} \binom{t/2+r-1}{t/2-1} \binom{t/2+n-r-1}{t/2-1} \quad (5.4.1)
 \end{aligned}$$

where  $m = n/2$ ,  $\delta_n = 1$  when  $n$  is even

and  $m = (n-1)/2$ ,  $\delta_n = 0$  when  $n$  is odd.

Using (2.1) and (2.2) this may be rearranged to yield:

$$D(t, n+1) = nt + \sum_{l=1}^k 2^{r-1} D(t/2^{r-1}, n) + \frac{t-1}{n+1} \binom{t+n-1}{n} + \sum_{l=1}^k 2^{r-1} \binom{t/2^{r-1} + (n-1)/2}{\lfloor (n+1)/2 \rfloor} \quad (5.4.2)$$

If we let  $m = \lfloor (n+1)/2 \rfloor$ , the last term of the right-hand side simplifies to

$$\frac{t}{2m!} \sum_{i=0}^{m-1} \sigma_i \cdot \frac{t^{m-i-1}}{2^{m-i-1}-1} \quad (5.4.3)$$

where  $\sigma_i$  is the  $i^{\text{th}}$  symmetric function on  $1, 2, \dots, m-1$ . We assume, after inspection of the first few special cases, that the general form for  $D(t, n)$  is

$$D(t, n) = \sum_{i=1}^n a_i t^{n-i+1} + t \sum_{i=n+1}^{2n-1} a_i k^{i-n} \quad (5.4.4)$$

where  $a_i = a_i^{(n)}$ . We have

$$\begin{aligned} \sum_{l=1}^k 2^{r-1} D(t/2^{r-1}, n) &= t \sum_{j=1}^{n-1} a_j 2^{n-j} \frac{(t^{n-j-1})}{(2^{n-j-1})} + \\ &+ t \sum_{j=n}^{2n-1} a_j \sum_{s=1}^k s^{j-n} \end{aligned} \quad (5.4.5)$$

Also

$$\begin{aligned} \frac{t-1}{n+1} \binom{t+n-1}{n} &= \frac{t^{n+1}}{(n+1)!} - \frac{t}{n(n+1)} + \frac{t}{(n+1)!} \sum_{j=1}^{n-1} (\sigma_j^{(n-1)} - \sigma_{j-1}^{(n-1)}) t^{n-j} \end{aligned} \quad (5.4.6)$$

Since algorithm BIND, in its leading terms, is the optimal binomial-expansion algorithm which uses recursion for computing powers of subpolynomials, it is worth quoting the closed form of the cost function  $D(t,n)$  in full detail. Substituting the previous four results in (5.4.2) and rearranging gives finally

$$\begin{aligned}
 D(t,n+1) = & \frac{t^{n+1}}{(n+1)!} + t \left[ n - \frac{1}{n(n+1)} - \frac{1}{2m!} \sum_{r=1}^{m-1} \sigma_{r-1}^{(m-1)} / (2^{m-r-1}) - \right. \\
 & - \sum_{j=1}^{n-1} a_j 2^{n-j} / (2^{n-j-1}) \left. \right] + \sum_{n-m+1}^{n-1} t^{n-r+1} \left[ \frac{\sigma_{r+m-n-1}^{(m-1)}}{2m! (2^{n-r-1})} + \right. \\
 & + \frac{a_r 2^{n-r}}{(2^{n-r-1})} + \frac{\sigma_r^{(n-1)} - \sigma_{r-1}^{(n-1)}}{(n+1)!} \left. \right] + \sum_{l=1}^{n-m} t^{n-r+1} \left[ \frac{a_r 2^{n-r}}{2^{n-r-1}} + \right. \\
 & + \frac{\sigma_r^{(n-1)} - \sigma_{r-1}^{(n-1)}}{(n+1)!} \left. \right] + kt(a_n + \frac{1}{2}m) + \\
 & + t \sum_{n+1}^{2n-1} a_j \sum_{i=1}^{j-n} i! \{ \frac{j-n}{i} \} \frac{(k+1)}{i+1} \quad (5.4.7)
 \end{aligned}$$

This formula gives the coefficients  $a_j^{(n+1)}$  of  $D(t,n+1)$  in terms of the coefficients  $a_j^{(n)}$  of  $D(t,n)$ . Cf. [2].

When  $n = 2, 3, 4$ , we have, always assuming  $t = 2^k$ ,

$$D(t,2) = t^2/2 + t/2 + kt/2 \quad (5.4.8)$$

$$D(t,3) = t^3/6 + t^2 + 5t/6 + t(5k/4 + k^2/4) \quad (5.4.9)$$

$$D(t,4) = t^4/24 + 11t^3/36 + 53t^2/24 + 4t/9 + t(7k/4 + 3k^2/4 + k^3/12) \quad (5.4.10)$$

• In general

$$D(t,n) = \frac{t^n}{n!} + t^{n-1} \left[ \frac{1}{2(n-2)!} + \frac{1}{(n-1)!(2^{n-2}-1)} \right] + o(t^{n-2}) \quad (5.4.11)$$

The improvement of BIND over BINC is given by

$$C(t,n) - D(t,n) = t^{n-1} \cdot \frac{1}{(n-1)!(2^{n-1}-2)} + o(t^{n-2}) \quad (5.4.12)$$

Another interesting result is that, for  $n \geq 2$ ,

$$\lim_{t \rightarrow \infty} \frac{D(t,n) - L(t,n)}{L(t,n)} = 0 \quad (5.4.13)$$

while

$$\lim_{t \rightarrow \infty} \frac{B(t,n) - L(t,n)}{L(t,n)} = n/2^{n-1} \quad (5.4.14)$$

This last result places BINB in a different asymptotic class from the superior binomial-expansion algorithms developed here. Thus far, we have seen the first few members of a sequence of successively better and better algorithms. With

the exception of the 'S' in BINS, the identifying letters (algorithm names) have all been chosen so that the time-complexity of the named algorithm decreases as we move through the alphabet (BINA, BINB, BINC, BIND, BINE, BINF). There is no BING. As the obvious and then the less obvious and then the subtle improvements are incorporated, it becomes more and more difficult to improve on each successive algorithm. The size of the improvement decreases. BINC differs from BINB in the leading term of the cost function, BIND from BINC in the second leading term. To see clearly the significance of each improvement, we must give exactly the analytic cost function of the new algorithm, and then compare the coefficients of the leading terms against the previous best member of the sequence. After BINC, there are no more improvements in the leading term. BINE, however, is another improvement in the second leading term.

#### 5.5 ALGORITHM BINE (Binomial E)

This algorithm is specified by the design decisions: binomial expansion, even splitting, multilevel splitting, binary merge for computing subpolynomial powers, and smaller. More formally, we make the following description. As usual,

$$f = f_1 + f_2.$$

Description:

- (1) Create the binary term-group tree for the polynomial

f:

- (a) Place f at the root
- (b) Place  $f_1$ ,  $\text{size}(f_1) = \lceil \text{size}(f)/2 \rceil$ , in the left sub-node
- (c) Place  $f_2$  in the right sub-node
- (d) Repeat steps (b) and (c) for the subpolynomials until each term of f has been placed at one of the terminal nodes of the tree

- (2) For each term of the original polynomial f, compute all powers from 2 to n. This completes the processing of the terminal nodes.

- (3) For each strictly interior node, both of whose sub-nodes have already been processed, compute all powers from 2 to n according to the following scheme:

$$(\text{node})^r = (\text{left sub-node} + \text{right sub-node})^r$$

expanded binomially.

- (i) For  $s = 1$  to  $r-1$  do

- (a) Multiply  $\binom{r}{s}$  by whichever of  $(\text{left sub-node})^s$  and  $(\text{right sub-node})^{r-s}$  has fewer terms.

- (b) Multiply the result in (a) by the remaining factor.



(ii) Collect (left sub-node)<sup>r</sup> + (right sub-node)<sup>r</sup> +  
the products computed in (i):

- (4) Compute the  $n^{\text{th}}$  power of the root according to the previous scheme.

### Analysis:

The importance of this algorithm justifies giving the analysis in full detail. But for distribution, BINE is (conjecturally) the optimal sequential binomial-expansion algorithm. We will make use of the results and closed-form expressions in Chapter II, and lay the groundwork for the space-complexity analyses in Chapter VI. We begin by developing the concept of a 'power group triangle', which is a representation for the binomial expansions (via sub-nodes) of the powers from 1 to  $n$  of a node, the so-called power group of that node. The representation follows from the 'smaller' idea.

Consider a typical power group, i.e., set of binomial expansions for all powers from 1 to  $n$ , for the  $n$  in  $f^n$ . More precisely, the power group triangle for a node  $a + b$  is the graphical representation in triangular form of all binomial expansions required to compute all powers from 1 to  $n$  of that node, given all powers from 1 to  $n$  of the two sub-nodes.

$$\begin{array}{l}
 a^6 + b^6 + a^5.6b + b^5.6a + a^4.15b^2 + b^4.15a^2 + a^3.20b^3 \\
 a^5 + b^5 + a^4.5b + b^4.5a + a^3.10b^2 + b^3.10a^2 \\
 a^4 + b^4 + a^3.4b + b^3.4a + a^2.6b^2 \\
 a^3 + b^3 + a^2.3b + b^2.3a \\
 a^2 + b^2 + a.2b \\
 a + b
 \end{array}$$

FIG. 5.1 THE POWER GROUP TRIANGLE WHEN  $n = 6$

This is actually a partially-specified algorithm for obtaining all powers from 1 to  $n$  of the polynomial  $a + b$  assuming the availability (apart from binomial coefficients) of all powers from 1 to  $n$  of both subpolynomials. Because  $a$  is the left sub-node, and  $b$  is the right sub-node,  $\text{size}(a) \geq \text{size}(b)$ . The first two columns do not involve computation. In each of the remaining columns, the so-called product columns, care has been taken to group the binomial coefficient together with the smaller of the two polynomials in each cross product. When  $t = 2^k$ , this is rigorously exact. When  $t \neq 2^k$ , there will be a few isolated instances when a larger power of the right sub-node will have fewer terms than a smaller power of the left sub-node. We ignore such instances and maintain the power group triangle in its present form.

Our first analytic task is to evaluate  $BC(s,n)$ , the total binomial-coefficient work required to create the power group of a node whose sub-nodes are of size  $s$ . For instance,

$$BC(s,2) = s$$

$$BC(s,3) = 3s$$

$$BC(s,4) = \frac{1}{2}(11s+s^2)$$

$$BC(s,5) = \frac{1}{2}(17s+3s^2)$$

That is, we must evaluate the sum of sizes of polynomials which are multiplied by binomial coefficients.

Now

$$\begin{aligned} BC(s,2m) - BC(s,2m-2) &= 4 \sum_{j=1}^{n-1} \binom{s+j-1}{j} + \binom{s+m-1}{m} \\ &= 4 \left[ \binom{s+m-1}{m-1} - 1 \right] + \binom{s+m-1}{m} \quad (5.5.1) \end{aligned}$$

Similarly

$$BC(s,2m+1) - BC(s,2m-1) = 4 \left[ \binom{s+m-1}{m-1} - 1 \right] + 3 \binom{s+m-1}{m} \quad (5.5.2)$$

Therefore

$$BC(s,n) - BC(s,n-2) = 4 \left[ \binom{s+p-1}{p-1} - 1 \right] + M_n \cdot \binom{s+p-1}{p} \quad (5.5.3)$$

where

$$\begin{aligned} p &= \lfloor n/2 \rfloor \quad \text{and} \quad M_n = 1 \quad \text{when } n \text{ is even} \\ &= 3 \quad \text{when } n \text{ is odd} \end{aligned}$$

Let  $x$  be 2 or 3.  $BC(s,x) = M_n \binom{s+0}{1}$ . A difference scheme, counting down from  $n$  by twos, gives

$$BC(s,n) - BC(s,x) = \sum_{j=1}^{p-1} \{ 4 \binom{s+j}{j} - 1 \} + M_n \binom{s+j}{j+1} \quad (5.5.4)$$

Using (2.1) for the summations and adding  $BC(s,x)$  gives

$$BC(s,n) = 4 \left[ \binom{s+p}{p-1} - p \right] + M_n \left[ \binom{s+p}{p} - 1 \right] \quad (5.5.5)$$

A closed form for  $BC(s,n)$  is therefore:

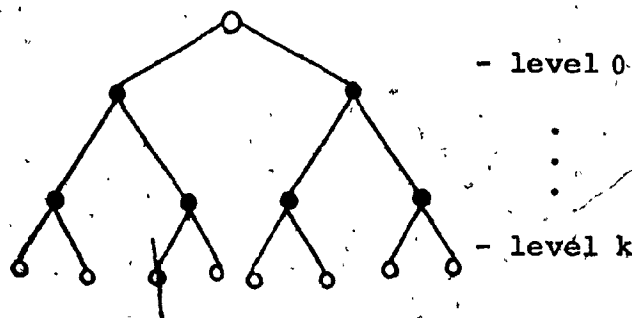
$$BC(s,n) = \sum_{j=1}^p \left\{ \frac{M_n}{p!} \binom{p+1}{j+1} \right\} + \frac{4}{(p-1)!} \sum_{k=0}^{p-j-1} \binom{p-1}{k+j} \binom{k+j}{k} 2^k s^j \quad (5.5.6)$$

where we have used (2.9) and (2.8), and where it is understood that  $\sum_{k=0}^{-1} \dots$  is zero. The leading term of  $BC(s,n)$  is  $\frac{M_n}{p!} s^p$ .

$BC(s,n)$  is a dense polynomial in  $s$ , of degree  $p$ , minus the constant term; this allows a convenient division of  $BC(s,n)$  by  $s$ .

In algorithm BINE, a power group is computed for each strictly interior node, i.e., each node neither terminal nor root. By summing over all interior nodes, we may obtain the total group binomial work, that is, the total number of multiplications by binomial coefficients involved in computing groups of powers.

Consider the following term group tree.



There is no binomial work at level  $k$ , the level of the terminal nodes. There is a power group triangle associated with each of the darkened, strictly interior nodes. The total binomial work here is (counting from the bottom):

$$\frac{t}{2} \cdot BC(1,n) + \frac{t}{4} \cdot BC(2,n) + \dots + 2 \cdot BC\left(\frac{t}{4}, n\right)$$

or

$$\frac{t}{2} \cdot \sum_{j=0}^{k-2} 2^{-j} BC(2^j, n)$$

Now

$$\frac{1}{s} BC(s,n) = \sum_{j=1}^p a_j^{(n)} s^{j-1} \quad (5.5.7)$$

where

$$a_j^{(n)} = \frac{M_n}{p!} [p+1]_{j+1} + \frac{4}{(p-1)!} \sum_{k=0}^{p-j-1} [p-1]_{k+j} \binom{k+j}{k} 2^k$$

The group binomial work (GBW) is, therefore,

$$GBW = \frac{t}{2} \cdot \sum_{m=0}^{k-2} \sum_{j=1}^p a_j^{(n)} (2^m)^{j-1}$$

$$= \frac{t}{2} \cdot \sum_{j=1}^p a_j^{(n)} \cdot \sum_{m=0}^{k-2} (2^m)^{j-1}$$

$$= \frac{t}{2} \cdot a_1^{(n)} (k-1) + \frac{t}{2} \cdot \sum_{j=1}^{p-1} a_{j+1}^{(n)} \frac{(t/2)^j - 1}{2^j - 1}$$

(5.5.8)

In addition to this group binomial work (GBW), there is root binomial work (RBW).

$$RBW = 2 \sum_{j=1}^{r-1} \binom{t/2+j-1}{j} + N_n \binom{t/2+r-1}{r} \quad (5.5.9)$$

where  $r = \lceil n/2 \rceil$  and  $N_n = 1$  when  $n$  is even  
 $= 0$  when  $n$  is odd.

Hence

$$\begin{aligned} RBW &= 2 \left[ \binom{t/2+r-1}{r-1} - 1 \right] + N_n \binom{t/2+r-1}{r} \\ &= 2 \cdot \text{group}(t/2, r-1) + N_n \cdot \text{size}(t/2, r) \\ &= \frac{2}{(r-1)!} \sum_{j=1}^{r-1} \binom{r}{j+1} \left(\frac{t}{2}\right)^j + \frac{N_n}{r!} \cdot \sum_{j=1}^r \binom{r}{j} \left(\frac{t}{2}\right)^j \quad (5.5.10) \end{aligned}$$

The total binomial work in algorithm binomial E (BINE) is the sum of the group binomial work and the root binomial work. The leading term of that work, when  $p > 1$ , is given by

$$\frac{1}{p!} \left(\frac{t}{2}\right)^p \cdot \left[ 2 - N_n + \frac{M_n}{2^{p-1}-1} \right] \quad (5.5.11)$$

where the first two terms are from RBW and the last term is from GBW. As the leading term of  $L(t, n)$  is  $t^n/n!$ , we see that the leading terms of  $E(t, n)$  result from non-binomial work. That is, since the binomial work does not affect the leading terms of  $E(t, n)$ , the differences in leading terms between  $L(t, n)$  and  $E(t, n)$  are due to non-binomial work.

The total non-binomial work in algorithm BINE is considerably easier to evaluate. It is the sum of the work required to process the terminal nodes and the work required to evaluate all cross products in all binomial expansions as if there were no binomial coefficients in binomial expansions. As before, we will split this non-binomial work into group work and root work. Consider again the term group tree.

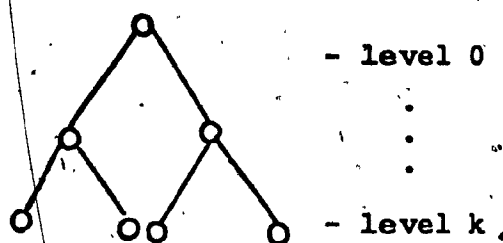


FIG. 5.3. A TERM GROUP TREE

The group work at level  $k$  is  $t(n-1)$ , which is

$$t.\text{group}(1,n) - t$$

If we write the binomial expansion with the binomial coefficients suppressed -

$$f^n = f_1^n + f_2^n + \sum f_1^r f_2^{n-r}$$

-, we see that the non-binomial work to compute  $f^n$  is given by  $\text{size}(f^n) - 2.\text{size}(f_1^n)$ , as there is one coefficient multiplication per extra term formed. Hence, after the group work at level  $k$  has been completed, the additional non-binomial

work required to compute all powers from 2 to n of all nodes at level k-1 is given by  $\frac{t}{2} \cdot \text{group}(2,n) - t \cdot \text{group}(1,n)$ . The total nonbinomial work to level k-1 is the sum

$$\frac{t}{2} \cdot \text{group}(2,n) - t$$

By continuing this argument, the total non-binomial work to level 1, the total group non-binomial work, is just

$$2 \cdot \text{group}(t/2,n) - t$$

The root non-binomial work is simply

$$\text{size}(t,n) - 2 \cdot \text{size}(t/2,n)$$

The total non-binomial work is their sum, namely,

$$\text{size}(t,n) + 2 \cdot \text{group}(t/2,n-1) - t$$

or

$$\frac{1}{n!} \sum_{j=1}^n \binom{n}{j} t^j + \frac{2}{(n-1)!} \sum_{j=1}^{n-1} \binom{n}{j+1} \left(\frac{t}{2}\right)^j - t \quad (5.5.12)$$

which is

$$\frac{t^n}{n!} + t^{n-1} \left[ \frac{1}{2(n-2)!} + \frac{1}{2^{n-2}(n-1)!} \right] + O(t^{n-2}) \quad (5.5.13)$$

This last result is extremely interesting for the following reason. We have seen that the leading terms of  $E(t,n)$  come from the non-binomial work. If the second term in brackets were zero, we would have the two leading terms the same as in



$L(t,n)$ . This means that, even if we reduce the binomial cost to zero, the second term in brackets shows an excess cost (compared to  $L(t,n)$ ) necessarily associated with a binary-merge, binomial-expansion algorithm. We can express that excess cost very simply. Subtracting  $L(t,n)$  from the total non-binomial work gives a non-binomial excess of  $2.\text{group}(t/2,n-1)$ . That is,

$$E(t,n) = L(t,n) + 2.\text{group}(t/2,n-1) + BW,$$

where  $BW$ , the total binomial work, is  $O(t^p)$ ,  $p = \lfloor n/2 \rfloor$ . The complete, closed-form expression for  $E(t,n)$  is the sum of the closed-form expressions for total non-binomial work and total binomial work, namely,

$$\begin{aligned} E(t,n) = & \frac{1}{n!} \sum_{j=1}^n \binom{n}{j} t^j + \frac{2}{(n-1)!} \sum_{j=1}^{n-1} \binom{n}{j+1} \left(\frac{t}{2}\right)^j - t + \\ & + \frac{2}{(r-1)!} \sum_{j=1}^{r-1} \binom{r}{j+1} \left(\frac{t}{2}\right)^j + \frac{N_n}{r!} \sum_{j=1}^r \binom{r}{j} \left(\frac{t}{2}\right)^j + \\ & + \frac{t}{2} \cdot a_1^{(n)} \cdot (k-1) + \frac{t}{2} \cdot \sum_{j=1}^{p-1} a_{j+1}^{(n)} \frac{(t/2)^{j-1}}{2^{j-1}} \quad (5.5.14) \end{aligned}$$

where

$$a_j^{(n)} = \frac{M_n}{p!} \binom{p+1}{j+1} + \frac{4}{(p-1)!} \sum_{k=0}^{p-j-1} \binom{p-1}{k+j} \binom{k+j}{k} 2^k$$

$$p = \lfloor n/2 \rfloor \quad N_n = 1, M_n = 1 \quad \text{when } n \text{ is even.}$$

$$r = \lfloor n/2 \rfloor \quad N_n = 0, M_n = 3 \quad \text{when } n \text{ is odd.}$$

We may use this to obtain  $E(t,n)$  for  $n = 2, 3, 4$ . We have:

$$E(t,2) = t^2/2 + t/2 + kt/2 \quad (5.5.15)$$

$$E(t,3) = t^3/6 + 3t^2/4 + t/3 + 3kt/2 \quad (5.5.16)$$

$$E(t,4) = t^4/24 + 7t^3/24 + 29t^2/24 - 2t/3 + 11kt/4 \quad (5.5.17)$$

The improvement of BINE over BIND may be expressed as

$$D(t,n) - E(t,n) = t^{n-1} \frac{1}{(n-1)! (2^{n-2}-1) 2^{n-2}} + o(t^{n-2}) \quad (5.5.18)$$

In the sequence  $B(t,n)$  to  $E(t,n)$  there is a strictly monotonic decrease both of the cost functions and of the differences between adjacent cost functions. The biggest improvement occurs for the BINB to BINC transition which shows up in the leading term of the cost function. BIND and BINE successively lower the coefficient of the second leading term. After BINE (i.e., BINF) the coefficient of the second leading term is not decreased. It is worth quoting again the two leading terms of  $E(t,n)$ , which, we conjecture, cannot be improved upon. They are not improved upon by BINF, even though BINF is an improvement. These terms are given by

$$E(t,n) = \frac{t^n}{n!} + t^{n-1} \left[ \frac{1}{2(n-2)!} + \frac{1}{2^{n-2}(n-1)!} \right] + o(t^{n-2}) \quad (5.5.19)$$

### 5.6 ALGORITHM BINF (Binomial F)

This algorithm is specified by the design decisions: binomial expansion, even splitting, multilevel splitting, modified merge for sub-polynomial powers, and smaller. BINF differs from BINE only in the handling of multiplication by binomial coefficients near the top of the term group tree. As such, it does not affect the non-binomial work, or the non-binomial-work cost functions, which are responsible for the leading terms of the total time-complexity cost function. The formal description is as follows. As before,  $f = f_1 + f_2$ .

#### Description:

- (1) Form the binary term group tree for the polynomial  $f$  in the usual manner; place  $f$  at the root, split  $f$  as evenly as possible placing the slightly larger half (if the sizes are not identical) in the left sub-node and the other half in the right sub-node, and continue this process until monomials are reached.
- (2) Process the terminal nodes (original monomials of  $f$ ) by forming all powers from 2 to  $n$ .
- (3) For all strictly interior nodes other than the two sons of the root, and both of whose sub-nodes have already been processed, compute all powers from 2 to  $n$  by:

$$(\text{node})^r = (\text{left sub-node} + \text{right sub-node})^r$$

expanded binomially.

(i) For  $s = 1$  to  $r-1$  do

(a) Multiply  $\binom{r}{s}$  by whichever of (left sub-node) $^s$  and (right sub-node) $^{r-s}$  has fewer terms.

(b) Multiply the result in (a) by the remaining factor.

(ii) Collect (left sub-node) $^r +$  (right sub-node) $^r +$  the products computed in (i).

(4) For the left and right sub-nodes of the root compute all powers from 2 to  $n$  except for the following:

(a) For the left sub-node, all powers from 2 to  $\lfloor (n-1)/2 \rfloor$ , if any

(b) For the right sub-node, all powers from 2 to  $\lceil (n-1)/2 \rceil$ , if any

(5) Compute the  $n^{\text{th}}$  power of the root according to the following scheme. Use binomial expansion in the manner of (3), that is, form each cross product of the expansion as (larger sub-polynomial power . (binomial coefficient . smaller sub-polynomial power)). If the smaller sub-polynomial has already been computed in (4), compute the inner parenthesis as indicated. Otherwise, compute the inner parenthesis by distributing the binomial coefficient over the summation which forms the binomial expansion of the

smaller sub-polynomial power. That is, if  $g^r$  is the smaller sub-polynomial power, compute  $\binom{n}{r} g^r$  as

$$\binom{n}{r} g_1^r + \binom{n}{r} g_2^r + \sum_s \binom{n}{r} \binom{r}{s} g_1^s g_2^{r-s}$$

where

$$f = g + h \quad \text{and} \quad g = g_1 + g_2$$

### Analysis:

We are interested in evaluating  $E(t,n) - F(t,n)$ , the number of binomial coefficient multiplications, if any, saved by not computing all powers from 2 to  $n$  of the left and right sub-nodes of the root. This savings of binomial coefficient multiplications accounts for all of the difference in time-complexity between BINE and BINF. The use of distribution to by-pass the independent computation of subpolynomial powers effectively reduces the amount of root binomial work; the most direct way to compute  $F(t,n)$ , then, is simply to re-evaluate the function  $RBW(t,n)$ . We have seen previously that, in BINE, this function is given by

$$\begin{aligned} RBW &= 2 \left[ \binom{t/2+r-1}{r-1} - 1 \right] + N_n \binom{t/2+r-1}{r} \\ &= 2 \cdot \text{group}(t/2, r-1) + N_n \cdot \text{size}(t/2, r) \quad (5.6.1) \end{aligned}$$

where

$$\begin{aligned} r &= \lfloor n/2 \rfloor \quad \text{and} \quad N_n = 1 \quad \text{when } n \text{ is even} \\ &= 0 \quad \text{when } n \text{ is odd} \end{aligned}$$

In BINF, distribution is used to evaluate products of the form  $\binom{n}{s} g^s$  in precisely two cases: (a) when  $g$  is the left sub-node of the root, and  $s$  lies between 2 and  $\lfloor (n-1)/2 \rfloor$ , and (b) when  $g$  is the right sub-node of the root, and  $s$  lies between 2 and  $\lceil (n-1)/2 \rceil$ . If  $g = g_1 + g_2$ , then distribution means: compute  $\binom{n}{s} g^s$  as

$$\binom{n}{s} g_1^s + \binom{n}{s} g_2^s + \sum_{j=1}^{s-1} \binom{n}{s} \binom{s}{j} g_1^j g_2^{s-j}$$

That is, if we allow for the cost of computing the  $\binom{n}{s} \binom{s}{j}$ , we avoid the cost of multiplying  $\binom{n}{s}$  by the product terms in the binomial expansion of  $g^s$ . The cost of forming the  $\binom{n}{s} \binom{s}{j}$  is  $\lfloor s/2 \rfloor$ , the number of distinct  $\binom{s}{j}$  when  $1 \leq j \leq s-1$ ; the resultant savings is that the binomial cost of computing  $\binom{n}{s} g^s$  drops from  $\text{size}(g^s)$  to  $2 \cdot \text{size}(g_1^s)$ . (The cost of multiplying the  $\binom{n}{s} \binom{s}{j}$  by the appropriate polynomials is already accounted for in the binomial cost of computing  $g^s$ , i.e., in GBW.) We note that, when  $s = 1$ , there are no product terms in the binomial expansion of  $g^s$ . Thus where, for BINE, RBW was given by

$$\text{RBW} = 2 \cdot \sum_{j=1}^{r-1} \binom{t/2+j-1}{j} + N_n \cdot \binom{t/2+r-1}{r} \quad (5.6.2)$$

it is now, for BINF, given by

$t = 2^k$ . Construct the sequences  $S_i$ ,  $1 \leq i \leq t$ , whose  $j^{\text{th}}$  term,  $1 \leq j \leq \text{size}(t, r-1)$ , is obtained by multiplying the  $i^{\text{th}}$  term of  $f$  by the  $j^{\text{th}}$  term of  $f^{r-1}$ . That is,  $S_i$  is the product of  $f^{r-1}$  with the  $i^{\text{th}}$  term of  $f$ . Next, merge  $S_{2i-1}$  with  $S_{2i}$  for  $1 \leq i \leq t/2$ , combining terms. Then merge the resulting sequences in pairs, combining terms, until one sorted sequence remains. In theory, the sorting time, namely,

$$O(t.k.\text{size}(t, r-1))$$

dominates the multiplication cost, namely,  $t.\text{size}(t, r-1)$ , but we are ignoring sorting time both for the sake of uniformity and to be generous to both RMUL and hence BINB. (Fateman asserts that the sorting time for any practical problem appears to be negligible [4]). None of our algorithms requires any sorting whatsoever, when the polynomials are completely sparse.

The largest core requirements for computing  $f^n$  by repeated multiplication occur during the step  $f^n = f.f^{n-1}$ . If all of the subsequences are obtained before merging, then  $t.\text{size}(t, n-1)$  terms need to be stored. This is a merge sort of  $t.\text{size}(t, n-1)$  numbers with  $t$  runs. Alternatively, if each subsequence is merged immediately after its formation, we are interested in the space required to merge the last subsequence with the sequence which is the union (by merge sort) of all previous subsequences. A merge sort of  $n$  records typically requires  $2n$  locations. We can avoid this, and sort in place, by applying a list merge sort (rearranging pointers) to polynomials represented as linked lists. The combined size

of the last two lists to be merged does not greatly exceed  $\text{size}(t,n)$ , the size of the final result. Given the diversity of multiplication-plus-sorting schemes, it is simplest to assign the following space complexity to RMUL, which is, in fact, an extremely generous lower limit. Taking the combined list size at the end as roughly  $\text{size}(t,n)$ , we assign a space complexity of  $\text{size}(t,n)$ , provided a link field is attached to each term. In that case, if  $\text{size}(f) = t$ , the space complexity of computing  $f^n$  by repeated multiplication is given by  $\text{size}(t,n) \cdot (1+E+P)$  central memory words.

It is our general conclusion that a linked-list representation for polynomials, with a storage requirement of  $(1+E+P)$  central-memory words per polynomial term, makes good sense from both the time-complexity and space-complexity stand-points, and this for all the sequential binomial-expansion algorithms considered in this thesis. If this be so, we need only measure the space complexity as number of terms of storage required. The final answer,  $f^n$ , may reasonably be written to disc, or some other form of secondary storage. Our real concern, then, is to determine, for each algorithm, the indispensable minimum core-storage required by the computation. This working storage will be essentially the space to store intermediate results after they have been obtained but before they have been used. The different ways in which the various algorithms generate and use intermediate results naturally give rise to different space complexities. As binary merge is a streamlined form of recursion (very much in the spirit, actually, of dynamic programming) in which pre-



cisely the minimum amount of intermediate results is generated, we concentrate here on comparing the space complexity of the algorithms which use merge for subpolynomial powering with the space complexity of the algorithms which use repeated multiplication. A family of implementation strategies for BINE will be considered, leading ultimately to a new algorithm, BINF.

#### 6.1 SPACE ANALYSIS FOR BINA AND BINB (Binomial A and Binomial B)

We consider in-core, linked-list implementations of these two algorithms, where the final answer,  $f^n$ , may be written to disc, and calculate the minimum storage required to store the intermediate results. The space complexities of BINA and BINB have never been analysed; in the case of BINB, we are faced with decisions concerning the implementation strategy which will radically affect the space complexity of the algorithm. In these algorithms,  $f^n$  is computed as

$$f_1^n + f_2^n + \sum_{r=1}^{n-1} f_1^r f_2^{n-r}$$

where the powers of  $f_1$  and  $f_2$  are first computed by repeated multiplication. The difference is that, in BINA,  $\text{size}(f_1) = 1$  and  $\text{size}(f_2) = t-1$ , while in BINB,  $\text{size}(f_1) = \text{size}(f_2) = t/2$ . In the first algorithm, i.e., BINA, the space required to compute  $f_2^n$ , dominates all other storage requirements. Using the previous generous lower limit for the space complexity of RMUL, we may say that the space complexity of algorithm BINA is  $\text{size}(t-1, n)$  terms. This may also be written

$$s_A = \frac{1}{n!} \sum_{j=1}^n \binom{n}{j} (t-1)^j$$

$$= \frac{t^n}{n!} + \frac{1}{n!} \sum_{j=1}^{n-1} (\binom{n-1}{j-1} - \binom{n-1}{j}) t^j \quad (6.1.1)$$

The coefficient of  $t^{n-1}$  is

$$\frac{1}{n!} \left[ \binom{n-1}{2} - 1 \right]$$

this space complexity is not radically less than  $\text{size}(t, n)$ , the size of the final answer.

That the space complexity of BINA is  $\text{size}(t-1, n)$  may be seen as follows. Because the sizes of the  $f_1^r$  are insignificant, one need not worry which subpolynomial powers to keep in core, nor in what order; one first generates all the  $\binom{n}{r} f_1^r$ , stores them, and then generates the successive powers of  $f_2$  (from 2 to  $n$ ) using each power in the binomial expansion as soon as it has been generated. In a word, the tasks of computing the  $\binom{n}{r} f_1^r f_2^{n-r}$  are independent subtasks. The most space is required to generate  $f_2^n$ , as explained. Yet no two powers of  $f_2$  (apart from  $f_2$  itself, of course) need be present in core at the same time. (One is assuming sophisticated garbage collection here, and the ability to overwrite  $f^n$  in the cells which held  $f^{n-1}$ .) Thus, ideally, one can manage with only  $\text{size}(t-1, n)$  cells, since  $O(t)$  cells more is negligible. In BINB, on the other hand, the sizes of  $f_1$  and  $f_2$ , and of their respective powers, are more or less the same. One must choose which powers to store in core, and this is not an easy choice.

If one attempts to hold all the powers of both  $f_1$  and  $f_2$  in core before expanding binomially, the total space required is  $2 \cdot \text{group}(t/2, n)$ , that is, twice the sum of sizes of all powers from 1 to  $n$  of a polynomial of size  $t/2$ . This is

$$\frac{2}{n!} \sum_{j=1}^n \binom{n+1}{j+1} \left(\frac{t}{2}\right)^j \quad (6.1.2)$$

which is not small, but much less than  $\text{size}(t, n)$ .

Two times  $\text{group}(t/2, n)$  is not a lower bound for the space complexity of BINB: we can do better. We observe first that a power of  $f_1$  or  $f_2$  not needed to generate higher powers, and already used in the binomial expansion of  $f^n$ , need not be retained in core. The non-negligible sizes of both the  $f_1^r$  and the  $f_2^r$  give us less flexibility here, but the following approach may be tried. (We maintain the, perhaps overgenerous, assumption that  $f_1^r = f_1 \cdot f_1^{r-1}$  may be computed in space size  $(t/2, r)$ .) First, generate all powers from 2 to  $p$  of both  $f_1$  and  $f_2$ , where  $p = \lfloor n/2 \rfloor$ . If  $n$  is even, the product  $\binom{n}{p} f_1^p f_2^p$  may now be formed, and  $p^{\text{th}}$  powers are no longer required in the binomial expansion. Next, successively generate the powers from  $p+1$  to  $n$  of  $f_1$ , using each power in the expansion as soon as generated (one uses  $f_1^n$  by writing it to disc), releasing powers of  $f_2$  whenever possible (once they have been used), and not retaining powers of  $f_1$  beyond  $f_1^p$ . Finally, successively generate the powers from  $p+1$  to  $n$  of  $f_2$ , proceeding in exactly the same manner. The least value of the minimum space required

occurs when  $n$  is even. After forming  $\binom{n}{p} f_1^p f_2^p$ , one needs to retain the powers from 1 to  $p-1$  of  $f_1$  (to match the as yet ungenerated higher powers of  $f_2$ ), and  $f_2^p$  (to generate these higher powers.) One also requires space to generate  $f_1^n$ . Thus, at best, the space complexity of algorithm BINB is  $\text{size}(t/2, n) + \text{group}(t/2, p)$ . This may also be written

$$S_B = \frac{1}{n!} \sum_{j=1}^n \binom{n}{j} \left(\frac{t}{2}\right)^j + \frac{1}{p!} \sum_{j=1}^p \binom{p+1}{j+1} \left(\frac{t}{2}\right)^j, \\ p = \lfloor n/2 \rfloor \quad (6.1.3)$$

For future reference, we list the leading terms corresponding to  $\text{size}(t, n)$  and to the space complexity functions obtained so far.

$$\text{size}(t, n) = \frac{t^n}{n!} + o(t^{n-1}) \quad (6.1.4)$$

$$S_A = \frac{t^n}{n!} + o(t^{n-1}) \quad (6.1.5)$$

$$2.\text{group}(t/2, n) = \frac{t^n}{n! \cdot 2^{n-1}} + o(t^{n-1}) \quad (6.1.6)$$

$$S_B = \frac{t^n}{n! 2^n} + o(t^{n-1}) \quad (6.1.7)$$

These functions are listed in strictly decreasing order. One sees that algorithm BINB has an impressively low space complexity, essentially that of the size of its single largest intermediate result,  $f_1^n$ . (The rest is asymptotically negligible.)

To attempt to match the BINB space complexity, a family of

implementation strategies for BINE will be considered, with the same time complexity but better and better space complexities, until finally BINF is considered, needing less time and requiring significantly less space. BINF was developed for space reasons. Of all known sequential algorithms for this problem, BINF has the least time and the least space.

## 6.2 SPACE ANALYSIS FOR BINE AND BINF (Binomial E and Binomial F)

We begin by establishing a result which we have tacitly assumed up until now, namely, if  $f_1^r$  and  $f_2^{n-r}$  are present in core then  $\binom{n}{r} f_1^r f_2^{n-r}$  may be obtained with space complexity not exceeding the space, if any, required to store the result. (That is, there is no need for working storage.) We form the product by retrieving each term of the smaller polynomial, multiplying by the binomial coefficient, and then retrieving and multiplying by each term of the larger polynomial, from which the result follows. Next, we consider the space required to store all powers from 1 to  $n$  of all subpolynomials associated with nodes of the multilevel term-group tree; this is potentially a rather large number. Consider a father node,  $f$ , and the two subnodes,  $f_1$  and  $f_2$ . Suppose  $f_1^n$  and  $f_2^n$  are present in core. We compute  $f^n$  according to

$$f^n = f_1^n + f_2^n + \sum \binom{n}{r} f_1^r f_2^{n-r}$$

The additional space required to store  $f^n$  is the space for the terms from the cross products. Suppose now the groups of

powers of all nodes at level  $k$ , are present in core. The additional space required to store the groups of powers of all nodes at level  $k-1$  is the space for the terms from the cross products. Continuing this argument, we may form all powers from 1 to  $n$  of all nodes at all levels from  $k$  to 1 (level 1 corresponds to the two subnodes of the root) until finally our space requirements increase to  $2.\text{group}(t/2, n)$ .

The linked-list representation for polynomials allows us to hold all of level  $k$  in core, and then all of level  $k-1$ , and so on, up to level 1, the level of the two sub-nodes of the root, without any garbage collection. This is because any power of any node at level  $j$  belongs to the binomial expansion of the same power of some node at level  $j-1$ . For example, if

$$g^r = g_1^r + g_2^r + \sum \binom{r}{s} g_1^s g_2^{r-s}$$

the linked list for  $g^r$  contains, first, the terms from the cross products, and second, the two previously existing lists,  $g_1^r$  and  $g_2^r$ . List concatenation absorbs the lists at a level into the lists at the next higher level. As the levels of the term-group tree are successively constructed (starting from the terminal nodes), no space is ever released; rather, with each new level, more space must be allocated for the new terms required to form that level. The most space required is that for the highest level formed, here,  $2.\text{group}(t/2, n)$ . Thus, a naive approach to implementing BINE yields exactly the same space complexity as the naive approach to implementing BINB,

except here there is no generous lower limit, sophisticated sorting, overwriting and garbage collection, just straight computation. Dynamic programming is a kind of recursion in which one keeps track of subproblems and never solves the same problem twice; the term-group tree, with its groups of powers of subpolynomials, is the table of solutions to all subproblems. The simplicity of the space complexity comes about because the solutions to the smaller subproblems are part of the solutions to the larger subproblems.

We have agreed that the final answer,  $f^n$ , may be written to disc rather than retained in core. Thus, once the groups of powers of the root subnodes have been obtained, there is no need for additional central memory. By retaining a full power group for each node other than the root, and writing the  $n^{\text{th}}$  power of the root out onto disc, we require  $2 \cdot \text{group}(t/2, n)$  terms of storage. Yet there is no need ever to retain the  $n^{\text{th}}$  power of any subpolynomial. Let the two subnodes of a node  $g$  be  $g_1$  and  $g_2$ .

$$g^n = g_1^n + g_2^n + \sum \binom{n}{r} g_1^r g_2^{n-r}$$

Suppose that  $g_1^n$  and  $g_2^n$  have been written to disc, and that all other powers of  $g_1$  and  $g_2$  are available in core. By writing the requisite cross products  $\binom{n}{r} g_1^r g_2^{n-r}$  to disc, we have written  $g^n$  to disc. We may in fact, write to disc the  $n^{\text{th}}$  power of every node, including in core only the 'limited' power group for each node, i.e., always excluding

power  $n$ . Not storing the  $n^{\text{th}}$  powers at level 1 (an obvious first thought) gives BINE a space complexity of  $2.\text{group}(t/2, n-1) + 4.\text{size}(t/4, n)$ ; not storing the  $n^{\text{th}}$  powers at any level gives BINE a much better space complexity, namely,  $2.\text{group}(t/2, n-1)$ . This may also be written

$$S_E = \frac{2}{(n-1)!} \sum_{j=1}^{n-1} \binom{n}{j+1} \left(\frac{t}{2}\right)^j \quad (6.2.1)$$

The leading term here is  $\frac{t^{n-1}}{(n-1)! 2^{n-2}}$ , and  $S_E < S_B$ , asymptotically.

So far we have discussed two sophisticated implementations of BINE, both with asymptotically smaller space complexities than the lower limit for the space complexity of BINB, both based on the idea of not retaining  $n^{\text{th}}$  powers of subpolynomials in core. BINB can make no use of this idea; in contrast to recursion or dynamic programming (here, binary merge), repeated multiplication is a whole polynomial method committed to building up  $f_1^n$  and  $f_2^n$  in core. To see the relative magnitudes of the space complexity functions for BINB (lower limit) and BINE (two implementations), consider the following three expressions.

$$S_B = \frac{1}{n!} \sum_{j=1}^n \binom{n}{j} \left(\frac{t}{2}\right)^j + \frac{1}{p!} \sum_{j=1}^p \binom{p+1}{j+1} \left(\frac{t}{2}\right)^j,$$

$$p = \lfloor n/2 \rfloor \quad (6.2.2)$$



$$2.\text{group}(t/2, n-1) + 4.\text{size}(t/4, n) = \frac{2}{(n-1)!} \sum_{j=1}^{n-1} \binom{n}{j+1} \left(\frac{t}{2}\right)^j + \frac{4}{n!} \sum_{j=1}^n \binom{n}{j} \left(\frac{t}{4}\right)^j \quad (6.2.3)$$

$$S_E = \frac{2}{(n-1)!} \sum_{j=1}^{n-1} \binom{n}{j+1} \left(\frac{t}{2}\right)^j \quad (6.2.4)$$

These three expressions are listed in strictly decreasing asymptotic order. The first has leading term  $\frac{t^n}{n! 2^n}$ ; the third has leading term

$$\frac{t^{n-1}}{(n-1)! 2^{n-2}}$$

When  $t$  grows large, with  $n$  fixed,  $S_B$  grows faster than  $S_E$ . (The cross-over point for the leading terms occurs for  $t = 4n$ .) Still, there will be values of  $t$  and  $n$  for which the BINB lower limit is less than the BINE actual value. We need to improve the space complexity of our use of dynamic programming.

When discussing the lower limit for BINB space complexity, we saw that the tasks of computing the  $\binom{n}{r} f_1^r f_2^{n-r}$  are essentially independent subtasks. There is no particular reason, when performing one subtask, to store the intermediate results necessary to perform some other subtask. As always,

$$f^n = f_1^n + f_2^n + \sum_{r=1}^{n-1} \binom{n}{r} f_1^r f_2^{n-r}$$

where  $f$  is the root, and  $f_1$  and  $f_2$  the nodes at level 1. Both  $f_1^n$  and  $f_2^n$  have already been written to disc. We now take the decision to generate the powers of nodes at level 1 (assuming the existence of the powers of nodes at level 2) only as they are successively needed in the binomial expansion. When  $t$  is sufficiently large, the most space required will be that to generate  $\binom{n}{1} f_1^{n-1} f_2$  (or vice-versa). Taking into account the groups of powers at level 2, we obtain a space complexity  $\underline{S}_E$  of size  $(t/2, n-1) + 4 \cdot \text{group}(t/4, n-1)$ , using a bar to distinguish the new space complexity function. We may write this BINE space complexity as

$$\begin{aligned} \underline{S}_E = & \frac{1}{(n-1)!} \sum_{j=1}^{n-1} \binom{n-1}{j} \left(\frac{t}{2}\right)^j + \\ & + \frac{4}{(n-1)!} \sum_{j=1}^{n-1} \binom{n}{j+1} \left(\frac{t}{4}\right)^j \end{aligned} \quad (6.2.5)$$

Surely,  $\underline{S}_E = \text{size}(t/2, n-1) + 4 \cdot \text{group}(t/4, n-1)$ , the lowest obtainable space complexity for algorithm BINE, is less than, i.e., asymptotically less than

$$S_B = \text{size}(t/2, n) + \text{group}(t/2, p), p = \lfloor n/2 \rfloor$$

which is a lower bound on the space complexity of algorithm BINB. The first term of the sum dominates in both cases, and  $\text{size}(t/2, n) > \text{size}(t/2, n-1)$ . The leading term of  $\underline{S}_E$  is

$$\frac{t^{n-1}}{(n-1)!} \left[ \frac{1}{2^{n-1}} + \frac{1}{2^{2n-4}} \right]$$

It seems extremely difficult to avoid in-core storage for the powers from 1 to  $n-1$  of the nodes at level 2; these powers are used again and again at level 1: they should be computed once, and stored. An improvement at level 1 with respect to storage is, however, possible, if we make use of distribution to precompute directly products of the form  $\binom{n}{r} f_1^r$ , which is the characterizing idea of algorithm BINF. Consider writing (say)  $f^4$  as

$$f_1^4 + f_2^4 + f_1^3 \cdot 4f_2 + f_2^3 \cdot 4f_1 + f_1^2 \cdot 6f_2^2,$$

i.e., according to the smaller idea. Apart from  $f_1^4$  and  $f_2^4$ , which have already been written to disc, the binomial expansion of the desired power of the root consists of a number of cross products of the form, larger polynomial times binomial coefficient times smaller polynomial. The larger polynomial will be called the a-list; the product of the smaller polynomial by the binomial coefficient will be called the b-list. The sum of interest thus becomes

$$\sum_i a\text{-list}_i \cdot b\text{-list}_i$$

These new objects, namely, a-lists and b-lists, belong to level 1 of the term-group tree. We suppose that level 2 has already been processed, and that all powers from 1 to  $n-1$

of all sub-polynomials at level 2 are available in core. The claim is made that the  $n^{\text{th}}$  power of the root may be written to disc using only the additional amount of central memory required to store the largest b-list. This gives algorithm BINP a total space complexity of  $\text{size}(t/2, p) + 4 \cdot \text{group}(t/4, n-1)$ ,  $p = \lfloor n/2 \rfloor$ . This may also be written

$$S_F = \frac{1}{p!} \sum_{j=1}^p \binom{p}{j} \left(\frac{t}{2}\right)^j + \frac{4}{(n-1)!} \sum_{j=1}^{n-1} \binom{n}{j+1} \left(\frac{t}{4}\right)^j,$$

$$p = \lfloor n/2 \rfloor \quad (6.2.6)$$

The leading term of  $S_F$  is  $\frac{t^{n-1}}{(n-1)! 2^{2n-4}}$ . Comparing the leading term of  $S_B$ , viz.,

$$\frac{t^n}{n! 2^n}$$

we see that  $S_F$  is far superior. The claim made above may be substantiated as follows. Consider a-list<sub>i</sub>, b-list<sub>i</sub> for a particular value of  $i$ . By distribution of the binomial coefficient over the binomial expansion of the smaller polynomial, the b-list may be represented in terms of constants and powers of sub-polynomials available at level 2. Hence, it may be computed, and stored in core in a space able, by definition, to hold the largest b-list. That space is of size  $\text{size}(t/2, p)$ .

We continue the argument. One particular b-list is now available in core. The corresponding a-list possesses a

binomial expansion in terms of binomial coefficients and powers of subpolynomials available at level 2. This a-list may be computed at essentially no additional central memory cost. Let the a-list be  $g^r$ , and let the binomial expansion be

$$g^r = g_1^r + g_2^r + \sum_{s=1}^{r-1} \binom{r}{s} g_1^s g_2^{r-s}$$

For each term of  $g_1^r$  and  $g_2^r$ , multiply by each term of the b-list, and write this product to disc. For the cross products, a slightly different strategy is adopted. Each cross product has a larger polynomial and a smaller polynomial, as usual. For each term of the smaller polynomial, multiply by the binomial coefficient, then by each term of the larger polynomial, and finally by each term of the b-list. The products computed in the inner loop are written to disc. In pseudo-Pascal, this is:

```

for each term of small do
    temp := coefficient * small [i]
    for each term of large do
        for each term of b-list do
            write(temp*large [j]*b-list [k])

```

The whole loop, of course, is executed once for each cross product in the binomial expansion of the a-list in question.

We can catalogue the various space-complexity functions obtained so far. The lower bound on the BINB space complexity is  $\text{size}(t/2, n) + \text{group}(t/2, p)$ ,  $p = \lfloor n/2 \rfloor$ . The space complex-

ities of the various implementations of BINE are:  $2.\text{group}(t/2, n)$ ,  $2.\text{group}(t/2, n-1) + 4.\text{size}(t/4, n)$ ,  $2.\text{group}(t/2, n-1)$ , and  $\text{size}(t/2, n-1) + 4.\text{group}(t/4, n-1)$ . The space complexity of the essentially unique implementation of BINF is  $\text{size}(t/2, p) + 4.\text{group}(t/4, n-1)$ ,  $p = \lfloor n/2 \rfloor$ . The E and F complexities do represent the space that would be used by these implementations; the B complexity is a lower bound which is rather generous, especially for small values of  $n$  and  $t$ . Clearly, we want to compare the relative values of these functions. The E and F series, as written above, is strictly monotonically decreasing; each function is obtained from the previous by subtracting a strictly positive quantity. The claim was made above that the E and F implementations which did not retain  $n^{\text{th}}$  powers in core were asymptotically superior to B, essentially on the grounds that  $n^{\text{th}}$  powers of sub-polynomials eventually outgrow  $n-1^{\text{st}}$  powers. But there is a danger in all such asymptotic arguments that the asymptotically superior algorithm becomes superior just as we pass beyond the bounds of the practically computable. Hence, we shall now make a more careful comparison of  $S_B$  and  $S_F$ , the space complexities, respectively, of the most successful repeated-multiplication algorithm, and the most successful dynamic-programming algorithm.

When  $t = 2$ , not too surprisingly, the differences between binomial-expansion algorithm disappear. Here, there is no difference between single-level and multi-level, dynamic programming is repeated multiplication, and no power of a monomial is smaller than any other. All the formulas given so far for time

complexity and space complexity, with one exception, were derived under the implicit assumption that  $t$  was a power of two; when this is not the case, the formulas are only approximate. To see the relative magnitudes of  $S_B$  and  $S_F$  very clearly, these functions were tabulated for various values of  $t = 2^k$  and  $n$ . When  $t = 4$ , BINB requires less space; the storage for intermediate results, however, is less even than the space required to store the program. When  $t \geq 8$ , BINF requires less space; in the one case of equality ( $t = 8$ ,  $n = 3$ ), the BINB lower limit can be shown to be inapplicable. Examination of the tabulated results shows that, in fact, it is over the whole range of the practically computable (apart from the trivially small) that the space complexity of BINF is superior to that of BINB. For a small problem,  $t = 8$ ,  $n = 10$ ,  $S_B = 411$ ,  $S_F = 272$ ; for a large problem  $t = 32$ ,  $n = 5$ ,  $S_B = 15,656$ ,  $S_F = 2,112$ . One may conclude, therefore, that dynamic programming, coupled with intelligent memory management, leads to space improvements more dramatic even than those in the time domain. One is certainly very far from the classical idea of a space-time trade-off. (The Table is Appendix III.)

This concludes the discussion of the sequential binomial-expansion algorithms. The results obtained depend totally on the two models for the problem which have been adopted in this thesis, namely, the cost model and the computational model. Nothing extends the validity of these results to other models. The computational model has been that the input polynomials be multivariate polynomials completely or almost completely sparse

to power  $n$ ,  $n$  the power sought. Obviously not all polynomials are sparse, yet existing algebra systems often need to handle multivariate problems of a sparse nature. The algorithms treat almost sparse polynomials as if they were totally sparse. The cost model has been that the true run-time cost may accurately be measured by the number of coefficient multiplications used in the algorithm to generate the final result; this model has been justified above. Finally, the space complexity of an algorithm has been defined as the central memory required to store intermediate results in the best implementation of that algorithm; assuming a linked-list representation for polynomials, space complexities have been quoted in number of terms of storage required. A term may require several central memory words, the exact number of which is algorithm-independent. These, then, are the chief assumptions about the input polynomials, and about the ways to measure the time and space complexities of the algorithms.

Very definite conclusions have been reached about which algorithms have least time complexity, and which least space complexity (BINF, apparently, is optimal in both respects.) In addition, conclusions have been drawn about the desirability of various design options which exist in sequential binomial-expansion algorithms. These conclusions, too, are model-dependent. One example can be given. When the polynomials are completely sparse, clearly it is best to split the polynomials as evenly as possible. This reduces costs, as we have



seen. Yet nothing rules out the possibility that, for completely dense polynomials, the best choice would have been to split the polynomials as unevenly as possible. And so on for the other decisions. The final conclusion is this. In this section we have analysed a family of sequential binomial-expansion algorithms for symbolic computation of integer powers of completely or almost completely sparse polynomials. We have analysed the time and space complexities of these algorithms. By a series of refinements and improvements, based on the ideas of dynamic programming and intelligent memory management, we have arrived at an algorithm, algorithm BINP (pronounced: binomial P), which we believe and conjecture to be optimal for both time and space within the binomial-expansion family under the assumptions listed above. Possibly, it is the optimal way to power sparse polynomials.

CHAPTER VII  
PARALLEL ALGORITHMS

## CHAPTER VII

### PARALLEL ALGORITHMS

In the discussion of the sequential algorithms we saw that the theoretical lower limit on the number of coefficient multiplications required to compute the  $n^{\text{th}}$  power of an arbitrary polynomial was given by  $\text{size}(t, n) - t$ , which, for large  $t$  and large  $n$ , can become an extremely large number. This result puts limits on what is practically computable with a sequential architecture. Recent technological advances, in principle, make possible the production of very cost-effective high-performance computers which make parallel use of a large number of processors. Researchers have analysed various parallel computer architectures, and the problems of adapting sequential algorithms to parallel machines, in applications areas where enormous amounts of straight computation are required. A major lesson has been that special-purpose machines which have been adequately tailored to their applications area can perform spectacularly, even if these same machines do much less well when applied to problems they were never intended to solve. We consider two special-purpose parallel architectures, one multiprocessor and the other associative-processor, then examine the adaptations necessary to run variants of the best sequential binomial-expansion algorithms on these machines, and finally calculate the speed-up ratio obtained for each architecture-algorithm combination. The general conclusion is that integer powers of sparse

polynomials are well-suited to parallel computation, with the actual speed-up ratios approaching the theoretical ideal.

A basic division of parallel architectures [6] is into single instruction stream, multiple data stream systems, and multiple instruction stream, multiple data stream systems. (There are other types.) In the first category, only one instruction is executing at any one time (single control unit), yet may act on a whole set of data (multiple processing units); examples are array processors and associative processors. In the second category many instructions execute simultaneously (multiple control units) on different data (associated multiple processing units); essentially, we have a system of interconnected conventional processors. Of course, an architecture can be devised which lies somewhere between these two extremes. (The two extremes are having many control-less functional units, and having many general-purpose computers.) In the associative-processor architecture envisaged here, a large associative memory (in which data is addressed by tag rather than by address) will be coupled with a parallel processing array whose elements perform more or less the same computation simultaneously. In the multiprocessor architecture, there will be a central control section, capable of sophisticated processing, and a number of slave processors with extremely limited control capabilities. That is, the former system will essentially be SIMD (single instruction stream, multiple data stream,) while the other system will keep the deviations from the SIMD concept to an acceptable minimum.

In general, one cannot obtain good parallel algorithms by simple translation of existing serial algorithms. This is because a particular parallel architecture will be efficient only if the computation to be run on it has a particular form. If one is running on a SIMD machine, then the original computation must be broken up into many smaller subcomputations which are structurally identical but which may have different data values. When such a splitting is not possible, one is forced to use an MIMD machine, in which the subcomputations have different structures as well as different data values. Thus, in the parallel case, on the one hand, there is a definite algorithm-machine interdependence, and, on the other hand, starting with a given serial algorithm for parallel adaptation will place constraints on the way the computation may be broken up into subcomputations. In the sequential binomial-expansion algorithms, one multiplies constants times polynomials, and polynomials times other polynomials. The first operation is monomial times polynomial; the second operation is monomial times polynomial many times. The most elementary operation is simply monomial times monomial. One way to achieve parallelism, insofar as the elementary operations are completely independent, is to split a composite task, such as polynomial times polynomial, into sets of elementary sub-tasks. Another way to achieve parallelism, specific to algorithms which employ multi-level splitting, is to take advantage of the structurally-identical subcomputations associated with different nodes belonging to one particular level of the term-group tree. These

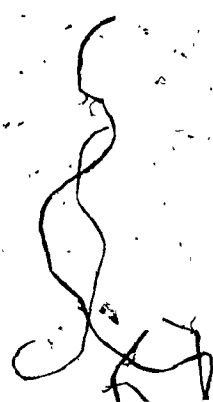
are the two main approaches. Both are used in connection with the multiprocessor architecture; the former alone is used in connection with the associative-processor architecture.

One needs a criterion to measure the success of a particular parallel algorithm-architecture combination, that is, essentially, some measure of whether the decrease in run-time warrants the additional expenditure for the parallel machine. Following Stone [14], we adopt the measure for specific problems of the speed-up ratio defined as:

$$\text{Speed-up ratio} = \frac{\text{Computation time on a serial computer}}{\text{Computation time on the parallel computer}}$$

To be fair, we compare the best serial algorithm for a specific problem with the best parallel algorithm for the same problem, whether or not the parallel algorithm is an adaptation of the serial algorithm. For a parallel architecture with a processor or functional multiplicity of  $N$ , the ideal speed-up ratio is  $N$ ; this is rarely obtainable. When the speed-up ratio is  $kN$ ,  $k < 1$ , but not much less, we have a problem very well-suited to parallel computation. Speed-up ratios of  $kN/\log_2 N$  are less desirable, while speed-up ratios of  $k \log_2 N$  are simply inadequate. In the present thesis, we create the best parallel algorithms by adapting the best sequential algorithms. As a rough approximation, we consider the cycle time of the parallel computers to be equal to that of the sequential machines. Hence, we measure our speed-up ratios as number of coefficient multiplications (which does not change from the sequential case)

divided by the number of cycles required to perform the computation on the parallel machine. For both parallel architectures considered here, the speed-up ratios approach the theoretical ideal.



CHAPTER VIII  
MULTIPROCESSOR OPTION  
(MULTIPROCESSOR E)



## CHAPTER VIII

### MULTIPROCESSOR OPTION (MULTIPROCESSOR E)

In this section we describe the parallel architecture envisaged, give a formal specification of the corresponding parallel algorithm, and obtain a lower bound on the speed-up ratio for this system. We choose to make a parallel adaptation of the sequential algorithm BINE, and call the resulting parallel algorithm: 'Multiprocessor E'. The basic ideas in adapting BINE are not radically different from those involved in adapting BINF; the description and analysis of Multiprocessor E, however, is somewhat simpler. In any case, ease of parallel adaptation is more important than the relatively minor time difference between BINE and BINF.

#### 8.1 DESCRIPTION OF THE ARCHITECTURE

The parallel machine consists of a control unit with the processing capabilities of a conventional computer,  $N$  processing units or slave processors capable of decoding a limited set of special-purpose instructions set into memory by the control unit, and a large random-access central memory to which both the control processor and all slave processors have full access. Scheduling and memory management are the job of the control processor, who allocates himself a reserved section of central memory. The slave processors read their instructions from other reserved sections of central memory, properly

initialized by the control processor. Let us take  $N$ , the number of slave processors, to be  $N = 2^8$ .

The block diagram of the system is:

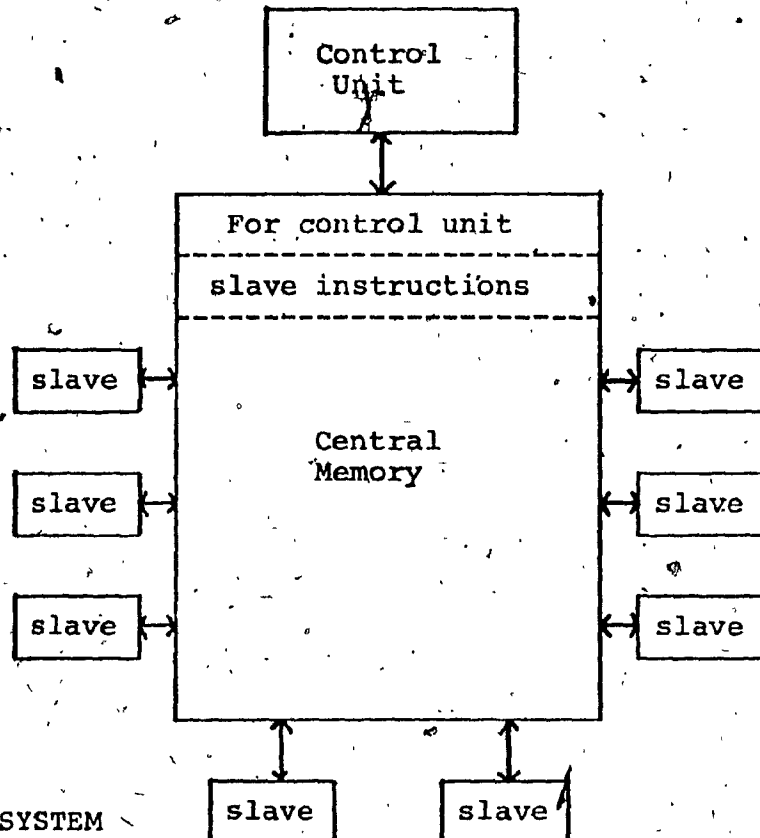


FIG. 8.1 SYSTEM DIAGRAM

The non-reserved portion of central memory is used for the storage of intermediate results generated by the computation, that is, by any of the slave processing units. Use of central memory is essentially additive, with more and more results being added to an initially empty store. This exploits the feature of dynamic programming in BINE in which solutions to smaller subproblems are parts of solutions to larger subproblems, as discussed above. One overwriting technique,

however, is employed: in the computation of  $g_1^r \cdot \binom{n}{r} g_2^{n-r}$ , the space ultimately allocated to hold the final result is initially allocated to hold  $\binom{n}{r} g_2^{n-r}$ . This allows a clean separation within the parallel algorithm of multiplication by binomial coefficients, and multiplication by the (remaining) polynomial.

We can give a rough idea of the kind of special-purpose instruction decoded by the slave processing units. Given the character of algorithm BINE, how might  $N$  independent processors cooperate to perform the overall computation? The latter consists of a number of list constant products, each of which, sequentially, is the multiplication of a list (polynomial) by a constant (a constant product) followed by the multiplication of the resulting list by another list (a list product.) A set of list constant products therefore, may be broken into a set of constant products, and a set of list products; we require that a processor may be set to do an arbitrary amount of the work required to compute a set of constant products, or a set of list products. These lists are, of course, real lists in the linked-list sense. A sublist of a list is itself a list. A list may be specified by giving both its lead element and its length. The special purpose instructions, or rather sets of special-purpose instructions, for the slave processing units should now be clear. One slave unit, as part of computing a set of constant products, might be instructed to multiply each of several lists by one of several constants. Another slave unit, as part of computing a set of list products, might be instructed to multiply each of several lists by

one of several other lists. Operands for these operations are specified more or less as described above; the target areas in central memory to which the final results are to be written must also be carefully specified. This specification is an overhead function assumed by the control unit.

The total computation of BINE consists of a number of sets of list constant products, one such set for each node of the term-group tree. Each set of list constant products consists of a set of constant products, and a set of list products. Each set of constant products, and each set of list products consists of a number of elementary (because monomial) and hence independent (multiplication) operations. The assumption that a slave processor may be set to do an arbitrary number of elementary operations implies that  $N$  independent slave processors may divide the total computation among themselves and yield a speed-up not radically different from  $N$ . We define the idea of an  $N$ -split of  $m$  subtasks  $S_i$ , where the subtask  $S_i$  consists of  $n_i$  elementary operations. Let

$$W = \sum n_i \quad \text{and} \quad Q = \lceil W/N \rceil$$

Create  $N$  (new) subtasks  $T_i$ , of maximum size  $Q$ , in the following way. Lay out the  $m$  (original) subtasks  $S_i$  in linear order. Count off  $Q$  elementary operations from the start of  $S_1$ ; this is the (artificial) subtask  $T_1$ . Continue in the same fashion to obtain the (artificial) subtasks  $T_2$  through  $T_N$ . Assign each of the  $N$  subtasks  $T_i$  to one of the  $N$

slave processors. This completes the N-split. In the present context, the  $m$  subtasks  $S_i$  are either  $m$  constant products, or  $m$  list products. That is, we make an N-split of either a set of constant products or a set of list products, and this some number of times until the total computation is completed. We now make this more formal.

The precise operation of a multiprocessor N-split may be explained as follows. For each value of  $n$ , there is an abstract object called the power group triangle, which is used in BINE to compute all powers from 1 to  $n$  of  $a + b$  given all powers from 1 to  $n$  of each of  $a$  and  $b$ . For example, when  $n = 4$ , the triangle is:

$$\begin{array}{l} a^4 + b^4 + a^3.4b + b^3.4a + a^2.6b^2 \\ a^3 + b^3 + a^2.3b + b^2.3a \\ a^2 + b^2 + a.2b \\ a + b \end{array}$$

FIG. 8.2 POWER GROUP TRIANGLE (pgt) FOR  $n = 4$

In a pgt, there are  $\binom{n}{2}$  constant products, and  $\binom{n}{2}$  list products. The number of elementary operations in each product is a computable function of  $s = \text{size}(a) = \text{size}(b)$ . That is, we have two sets of  $\binom{n}{2}$  subtasks  $S_i$ , where we can evaluate the  $n_i$  for each  $S_i$  by knowing  $s$ . That is, for each node in

the term-group tree, we are able to perform an N-split (slave processor allocation) for all the binomial work in the pgt, and then a second N-split for the remaining nonbinomial work. The  $n_i$  are polynomial functions of  $s$ , which are fixed once and for all for a given value of  $n$  (the  $n$  in  $f^n$ ), but which must be re-evaluated by the control unit each time the value of  $s$  (equivalently, tree level) changes. Once the values of the  $n_i$  are known, the N-split (slave processor allocation) proceeds as described above.

## 8.2 DESCRIPTION OF THE ALGORITHM

### Assumptions:

Let  $t$ , i.e.,  $\text{size}(f)$ , and  $N$ , i.e., the slave processor multiplicity, both be powers of two, with  $N \geq t$ . The changes to the algorithm when  $t > N$  are trivial.

### Step 1 - Creation of the Term-Group Tree

The control unit, in its section of central memory, creates a binary term-group tree for the original polynomial  $f$ . All of  $f$  goes into the root, the two halves of  $f$  go into the two subnodes of the root, and so on, recursively, with the larger half always in the left subnode. A directory is maintained which, as the computation proceeds, for each node other than the root, and each power from 1 to  $n$ , points to the list which contains the specified power of the specified subpolynomial.

## Step 2 - Processing of the Terminal Nodes

The control unit assigns one slave processor to each of the  $t$  terminal nodes, and lets the other  $N-t$  slave processors sit idle. All powers from 2 to  $n$  of each term of the original polynomial are computed. This completes the processing of the terminal nodes. The list pointers are automatically retained in the control-unit directory.

## Step 3 - Processing of the Interior Nodes

For each level of the term-group tree from  $k-1$  to 1, in that order, the control unit causes all powers from 2 to  $n$  of all nodes on that level to be computed, taking advantage of the powers already available on the next lower level. The fundamental strategy for computing a group of powers of a node, given the groups of powers of the subnodes, is expressed in the structure of the power group triangle, discussed previously. At level  $j$ , there are  $2^j$  nodes, with  $j < k$ . The control unit assigns  $M = N/2^j$  slave processors to each of the  $2^j$  nodes at level  $j$ .  $M$  slave processors cooperate to process one product group triangle, i.e., one node. The control unit evaluates the time complexity of the constant products and list products involved in processing nodes at this level. The control unit then causes each group of  $M$  slave processors associated with a node to perform, first an  $M$ -split of all binomial work in the pgt, and then an  $M$ -split of the (remaining) non-binomial work. That is, the  $M$  processors split the

constant products, and then the list products. The  $2^j$  groups of slave processors work in parallel. At the end of this step, all powers from 2 to  $n$  of all interior nodes will have been computed. As before, the list pointers are automatically retained in the control-unit directory.

#### Step 4 - Processing of the Root Node

At the root level, all  $N$  slave processors are assigned to compute  $f^n$ , given the availability of the powers from 1 to  $n$  of the two sub-polynomials  $f_1$  and  $f_2$ . Although only the  $n^{\text{th}}$  power of the root is computed, the same strategy may be applied. (We merely consider the binomial-expansion of the root, written in accord with the smaller idea, to be a degenerate pgt.) As before, then, the control unit causes the group of  $N$  slave processors to perform, first an  $N$ -split of the  $n-1$  constant products of the root pgt, and then an  $N$ -split of the  $n-1$  list products. This completes the processing of the root.

Multiprocessor E, like its sequential counterpart, BINE, makes full use of the design decisions: even splitting, multi-level splitting, binary merge (dynamic programming), and smaller. These are the decisions which fix the structure of the term-group tree, and of the product group triangle. The latter merely expresses how groups of powers of polynomials are to be computed from groups of powers of subpolynomials via binomial-expansion using the smaller idea; it is a kind of



abbreviated algorithm description. The term-group tree, a multilevel, even-splitting expansion of the original polynomial, serves as a directory into the various lists and sublists generated during the computation. The pgt's are processed by the slave processors under the control of the control unit, which also manages the term-group tree.

### 8.3 ANALYSIS OF THE SPEED-UP RATIO

If a subcomputation containing  $s$  independently-allocatable units of work be divided (p-split) among  $p$  independent processors, then the speed-up is given by  $s/[s/p]$ , where  $s$  measures the time for the serial computation, and  $[s/p]$  measures the longest time taken by any of the  $p$  co-operating processors. The speed-up is less than or equal to  $p$ . If two subcomputations, with  $s_1$  and  $s_2$  units of work, respectively, are successively p-split, then the speed-up is given by

$$\frac{s}{([s_1/p] + [s_2/p])}$$

where

$$s = s_1 + s_2$$

In general,

$$[s_1/p] + [s_2/p] \geq [s/p]$$

That is, in order to maximize the speed-up of the entire computation, one should minimize the number of sub-computations which

are p-split. Multiprocessor E uses variable p-splitting insofar as M (the processor multiplicity) is a function of tree level, ranging from 1 at level k to N at level 0. As the monomial products of the computation are independently allocatable, the speed-up ratio of Multiprocessor E is given by  $E(t,n)$  divided by the sum of the longest time taken each time a p-split occurs. The number and multiplicity of the p-splits are entirely fixed by the algorithm specification; they need only be summed, or at least approximated. This we now do.

More precisely, we calculate T, the time taken by algorithm Multiprocessor E, where T is measured in units of the time taken by a slave processor to compute one monomial product. For each non-terminal node, there are 2 p-splits: one for the constant products, and one for the list products. To calculate the time taken to process a node, we need to know the processor multiplicity, the number of monomial products in all constant products, and the number of monomial products in all list products. For example, the time taken to process the root is

$$\left[ \frac{2 \cdot \text{group}(t/2, r-1) + N_n \cdot \text{size}(t/2, r)}{N} \right] + \left[ \frac{\text{size}(t, n) - 2 \cdot \text{size}(t/2, n)}{N} \right] \quad (8.3)$$

where  $r = \lceil n/2 \rceil$ ,  $N_n = 1$  (n even) or 0 (n odd), and N is the number of slave processors available on the multiprocessor system. The time taken to process a strictly interior node is

$$\left\lceil \frac{BC(s,n)}{M} \right\rceil + \left\lceil \frac{\text{group}(2s,n) - 2.\text{group}(s,n)}{M} \right\rceil \quad (8.3.2)$$

where  $s$  is the subnode size, and  $M$  the processor multiplicity (per node) available at that level. If we add in the  $n-1$  time units needed to compute all powers from 2 to  $n$  of the terminal nodes (one processor per node), and take the sum over all levels containing strictly interior nodes, we obtain finally the total time for Multiprocessor E. It is

$$\begin{aligned} T(t,n) = & \left\lceil \frac{\text{size}(t,n) - 2.\text{size}(t/2,n)}{N} \right\rceil + \\ & + \left\lceil \frac{2.\text{group}(t/2,r-1) + N_n.\text{size}(t/2,r)}{N} \right\rceil + \\ & + \sum_{j=1}^{k-1} \left\{ \left\lceil \frac{\text{group}(t/2^j,n) - 2.\text{group}(t/2^{j+1},n)}{N/2^j} \right\rceil + \right. \\ & \left. + \left\lceil \frac{BC(t/2^{j+1},n)}{N/2^j} \right\rceil \right\} + n - 1 \end{aligned} \quad (8.3.3)$$

where, again  $r = \lceil n/2 \rceil$ ,  $N_n = 1$  ( $n$  even) or  $0$  ( $n$  odd), and  $N$  is the total number of slave processors in the system. The sum runs over levels rather than nodes because the groups of  $M$  processors (per node) run in parallel. The actual speed-up ratio is given by  $E(t,n)/T(t,n)$ . We use a simple trick to obtain a lower bound on this ratio by obtaining an upper bound on  $T(t,n)$ . In general,  $\lceil s/p \rceil < s/p + 1$ . The quantity in curly brackets is

$$\left\lceil \frac{Q_1^{(j)}}{N/2^j} \right\rceil + \left\lceil \frac{Q_2^{(j)}}{N/2^j} \right\rceil < \frac{2^j (Q_1^{(j)} + Q_2^{(j)})}{N} + 2 \quad (8.3.4)$$

But  $2^j (Q_1^{(j)} + Q_2^{(j)})$  is the number of coefficient multiplications which occur at level  $j$ . (The quantity in parentheses is the number per node.) The same reasoning applies at the root level, that is,

$$\left\lceil \frac{R_1}{N} \right\rceil + \left\lceil \frac{R_2}{N} \right\rceil < \frac{R_1 + R_2}{N} + 2 \quad (8.3.5)$$

Here,  $R_1 + R_2$  is the number of coefficient multiplications at the root level. The time taken to process the terminal nodes, namely,  $n-1$ , may be written as

$$n - 1 = \frac{t(n-1)}{N} + \frac{(N-t)(n-1)}{N} \quad (8.3.6)$$

Substituting the previous (in)equalities in the formula for  $T(t,n)$  gives

$$T(t,n) < \frac{E(t,n)}{N} + 2k + \frac{(N-t)(n-1)}{N} \quad (8.3.7)$$


or

$$\frac{E(t,n)}{T(t,n)} > N \cdot \frac{E(t,n)}{E(t,n) + 2kN + (N-t)(n-1)} \quad (8.3.8)$$

Take  $N = 2t$  for definiteness. For large  $t$  or large  $n$ ,  $t(4k+n-1)$  is negligible in comparison with  $E(t,n)$ . That is, for large  $t$  or large  $n$ ,  $E(t,n)/T(t,n)$ , the speed-up ratio

for algorithm Multiprocessor E, approaches the theoretical ideal of  $N$ , the number of processors. The only counterbalancing factor is the necessary overhead (assumed by the control unit) in properly instructing the slave processors. The most difficult and expensive operation here is probably determining the starting addresses of the various sublists assigned to the slave processors for processing.

As it now stands, algorithm Multiprocessor E does not have a space complexity as low as the more sophisticated implementations of the sequential algorithm BINE. An algorithm which consistently computes the constant products in a pgt before the list products must have space to store those constant products. This consistent approach, of course, allows a consistent and simplified p-splitting strategy. At the lower levels of the tree, one may write the constant products into the space which will subsequently contain the list products. At the lower levels, then, the space complexity does not change. At the higher levels, though, where the answers are written to disc directly, more storage will be required for the intermediate results. There is, however, no reason why a sophisticated version of Multiprocessor F could not approach the space complexity of the sequential algorithm BINP. One would simply require a more elaborate p-splitting strategy. Thus, we do not regard Multiprocessor E as in any way an ideal or optimal multiprocessor algorithm difficult to improve upon; it is merely a very clear indication how a



multiprocessor parallel architecture could be exploited to yield a dramatic speed-up of the sequential computation. There is no doubt, however, that Multiprocessor E could be much improved with respect to space complexity.

CHAPTER IX

ASSOCIATIVE-PROCESSOR OPTION  
(ASSOCIATIVE F)

## CHAPTER IX

### ASSOCIATIVE-PROCESSOR OPTION (ASSOCIATIVE F)

Once more, in this section we describe the parallel architecture envisaged, give a formal specification of the corresponding parallel algorithm, and obtain a lower bound on the speed-up ratio for the system. This time, however, we choose to make a parallel adaptation of the sequential algorithm BINP, and call the resulting parallel algorithm: 'Associative F'.

#### 9.1 DESCRIPTION OF THE ARCHITECTURE

The parallel machine consists of (1) a control unit with the processing capabilities of a conventional computer, (2) a large associative or content-addressable memory in which each memory cell contains both a tag field and a term (monomial) field, and retrieval is by tag value, (3) a moderately large term buffer for storing intermediate results, and finally, (4) a parallel processing array, that is, an array of processing elements, in which each processing element, properly initialized, computes one monomial product in parallel with all other processing elements. Each element of the parallel processing array (PPA) contains a tag field and two term fields,  $t_1$  and  $t_2$ . A term field may contain one term (monomial). When the PPA is fired, each element which has been loaded computes the product  $t_2 := t_1 * t_2$ ; the resulting



values of the  $t_2$  fields may be routed either to the term buffer or to the associative memory.

The block diagram for the simplest form of this system is as follows:

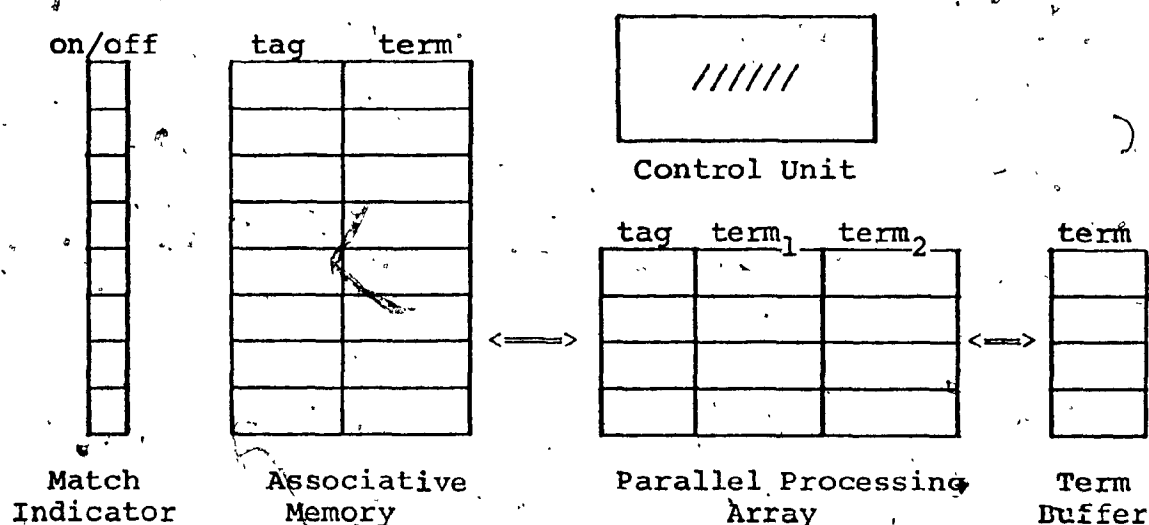


FIG. 9.1 SYSTEM DIAGRAM

The control unit is in communication with all other system units. When the control unit sets a tag value into the command buffer of the associative memory, the entire memory is searched in parallel and any matches of tag values are automatically recorded in the one-bit-per-cell match indicator. The control unit may cause the term fields of matched associative-memory cells to be routed to the  $term_1$  fields of the parallel processing array. The control unit initializes the  $term_2$  fields of the PPA either by routing terms from the term buffer or by inserting terms computed within the control unit itself. The control unit also initializes the tag fields of the PPA.

Roughly speaking, a tag value identifies a term belonging to a specific power of a specific sub-polynomial, that is, to one of the various intermediate polynomials generated during the computation. With an associative memory, rather than storing and retrieving these polynomials as explicitly-linked lists, we do so by storing and retrieving the terms belonging to a particular polynomial on the basis of their identifying tag values. Just as we can manipulate linked lists by resetting the pointers, so we can manipulate tagged lists by resetting the tags. The control unit (CU) requests a specific polynomial from the associative memory (AM) by giving it the appropriate tag value; the retrieved terms are then eventually routed to the  $t_1$  fields of the PPA, where they are used in the computation. The PPA generates new terms of new polynomials in the  $t_2$  fields. If the tag fields of the PPA have been properly initialized (by the CU) with the new tag values corresponding to the new polynomials, then, after the PPA has fired, the  $t_2$  fields of the PPA, together with the corresponding tag fields, may be routed back to the AM to store the new polynomials in the AM so that they can subsequently be retrieved for further computation. It is, of course, also the responsibility of the CU to initialize the  $t_2$  fields of the PPA prior to each firing. This is one of the two ways the PPA is used in Associative F.

Clearly, there is a certain overhead in storing intermediate results in the AM and then retrieving them later for further computation. Therefore, in certain circumstances, we

choose to route the new terms computed in the  $t_2$  fields of the PPA to the term buffer, without any corresponding tag values, and to retrieve them later by the simplest form of random access. PPA results go to the AM when terminal nodes are processed, or when list products are computed. The alternative routing is used during the computation of all the constant products in any one product column of a pgt. Consider, once more, a typical pgt, this one for  $n = 5$ .

$$\begin{array}{l}
 a^5 + b^5 + a^4.5b + b^4.5a + a^3.10b^2 + b^3.10a^2 \\
 a^4 + b^4 + a^3.4b + b^3.4a + a^2.6b^2 \\
 a^3 + b^3 + a^2.3b + b^2.3a \\
 a^2 + b^2 + a.2b \\
 a + b
 \end{array}$$

FIG. 9:2 Pgt FOR  $n = 5$

Our task is to up-date the AM, substituting the power group of the father node  $a + b$  for the power groups of the two sub-nodes  $a$  and  $b$ . The first two columns of the pgt are re-tagging columns, as this is all that is required here. The remaining columns are product columns, requiring the computation of new terms by multiplications performed within the PPA. For each product column, first the constant products are sent to the term buffer, and then the list products are sent to the

associative memory.

The simplest form of Multiprocessor E will run only if the corresponding parallel machine has a generous supply of central memory. Similarly, the simplest form of Associative F requires a large associative memory, and a relatively large term buffer. Both algorithms may be refined (and complicated) when the large space requirements of the simple forms become critical. In the simplest form of Associative F, the product columns of each pgt are processed separately; all constant products for a column are accumulated in the term buffer. Therefore, to process a node, we need buffer space to store the largest collection of constant products. Let  $s$  be the subnode size. When  $s$  and  $n$  are sufficiently large, we require a buffer able to hold size  $(s, p)$  terms,  $p = \lfloor n/2 \rfloor$ . For root processing,  $s = t/2$ . The associative memory must be able to hold the intermediate results from level 1 of the term-group tree; this amounts to  $2 \cdot \text{group}(t/2, n)$  associative-memory cells, and the ability to do parallel searches on associative memories of this size. We now present a version of Associative F which is suitable for this very large and powerful parallel machine. Again, we are more concerned with showing how to exploit an associative parallel architecture than with displaying the optimal associative algorithm; there is no doubt that improvements and refinements are possible.

We suppose that the parallel processing array has a multiplicity of  $N$ , and clarify the precise operation of a

PPA N-split. Consider one product column, and the constant products within it. One polynomial is to be multiplied by one or more binomial coefficients. The set of constant products is a set of subtasks in which  $N$  elementary operations at a time can be executed in parallel. As many copies of the polynomial as there are constant products are concatenated to form one long polynomial. The first  $N$  terms of this long polynomial are loaded into the  $N$   $t_1$ -fields of the PPA. The appropriate binomial coefficients are then loaded into the  $N$   $t_2$ -fields. The PPA is fired and the results are routed to the term buffer. This process is continued until all constant products for that product column have been computed. If  $W$  is the number of elementary operations in the set of constant products, then  $\lceil W/N \rceil$  firings of the PPA will be required to process the entire set. It is an important feature of constant-product N-splits that the PPA is fired precisely once for each time it is loaded. Once the set of constant products for a product column has been accumulated in the term buffer, the list products for that column may be computed. The list-product N-splits are slightly more complicated, basically in that the PPA is fired several times for each time that it is loaded. This will now also be explained.

The list products in a product column consist of, one or more times, a distinct polynomial times another distinct polynomial. These latter distinct polynomials are the constant products which have just been computed, and which are all of

precisely the same size. In a previous terminology, we have, one or more times, an a-list times a b-list. The a-lists are of varying sizes, and are available in the AM; the b-lists are all of the same size, and are available in the term buffer. All the a-lists of the product column are concatenated to form one long polynomial. The first  $N$  terms of this long polynomial are loaded into the  $N$   $t_1$ -fields of the PPA. The appropriate first terms of the various b-lists are then loaded into the  $N$   $t_2$ -fields. The PPA is fired and the results are routed to the AM. The process of loading the  $t_2$  fields and firing the PPA is continued until the b-lists are exhausted. (They will all be exhausted simultaneously.) The next  $N$  terms of the long polynomial are loaded into the  $t_1$  fields, and the whole process of  $t_1$ -loading,  $t_2$ -loading, and PPA-firing is continued until all list products for that product column have been computed. If  $W$  is the number of terms in the long polynomial, and  $S$  is the b-list size, then  $S \cdot \lceil W/N \rceil$  firings of the PPA will be required to process the entire set of list products for the product column. More exactly now, the PPA is fired precisely  $S$  times for each time the  $t_1$  fields of the PPA are loaded.

## 9.2 DESCRIPTION OF THE ALGORITHM

### Assumptions:

Let  $N$ , the PPA multiplicity, be greater than or equal to  $t$ , i.e., size (f). The changes to the algorithm when  $t > N$

are trivial. Moreover, let  $B$ , the buffer size, be sufficient to hold the largest collection of constant products of any product column. When this condition is not met, systematic modifications to Associative  $F$  can be made which allow the computation to go through, but which, naturally, reduce the speed-up ratio.

#### Step 1 - Creation of the Term-Group Tree

This is essentially a book-keeping operation. Using its own private random-access memory, the control unit creates the term-group tree, not by physically storing terms of the polynomial, but rather by allocating tag values which are sufficient to uniquely identify specific powers of specific subpolynomials. As in the multiprocessor case, the term-group tree functions as a directory into the various lists (polynomials) stored in the AM. The term-group tree essentially specifies the node-subnode relationship, and gives the tag values for all nodes and all powers concerned.

#### Step 2 - Processing of the Terminal Nodes

The control unit gates the  $t$  terms of the original polynomial into  $t$  of the  $t_2$  fields of the PPA.  $N-t$  processing units sit idle throughout this step. The corresponding tag values are initialized by the CU and, immediately, terms together with tag values are gated to the AM. Next, the  $t$  terms are copied into the matching  $t_1$ -fields of the PPA. The latter is fired  $n-1$  times ( $t_2 := t_1 * t_2$ ), creating

the powers from 2 to  $n$  of the terminal nodes. The tag values are initialized prior to each firing, and terms plus tags are gated to the AM after each firing. The PPA can send to the AM only from its  $t_2$  fields, and can receive from the AM only in its  $t_1$  fields. After  $n-1$  firings of the PPA, all terminal node processing is complete.

### Step 3 - Processing of the Lower Interior Nodes

For each level of the term-group tree from  $k-1$  to 2, in that order, all powers from 2 to  $n$  of all nodes at that level are computed. As Associative F uses distribution, level 1 is treated separately. The fundamental strategy for computing a group of powers of a node, given the groups of powers of the two sub-nodes, is as follows. The entire PPA is allocated to one node at a time. The pgt for that node is processed product column by product column, let us say from left to right. Each product column consists of some number  $m$  of products of the form a-list times b-list. The a-lists are distinct polynomials; the b-lists are distinct binomial coefficients times a unique polynomial associated with the product column. All b-lists for the column are computed and stored in the term buffer using the constant-product N-split described earlier; the unique polynomial is retrieved from the AM. Next the  $m$  list products in the column are computed and stored in the AM using the list-product N-split described in the same place; the  $m$  a-lists are retrieved from the AM as needed to load the  $t_1$  fields of the PPA. Prior to each firing of the PPA,



the CU initializes the tag fields with appropriate tag values, and the  $t_2$  fields with terms taken from the term buffer. After each firing, the new values of the  $t_2$  fields, and the corresponding values of the tag fields (which identify the lists to which the new terms belong) are routed to the AM. After all product columns of all nodes at a level  $j$  have been processed, the CU causes all terms at level  $j + 1$  to be retagged, making the lists at the lower level part of the new level, and thus completing the process of writing the power groups of all nodes at level  $j$  into the AM. At the end of step 3, the AM contains exclusively the power groups of all nodes at level 2.

#### Step 4 - Processing of the Nodes at Level 1

Processing a pgt at level 1 is structurally identical to processing a pgt at a lower level. The difference is precisely that, in some places, distribution coefficients are substituted for binomial coefficients before the computation begins. If a binomial expansion at level 1 will be used in a b-list at level 0, then all binomial coefficients in that expansion, if any, must be substituted. For example, suppose that  $a, b, c$ , and  $d$  are the nodes at level 2, and that  $(a+b)^2 \cdot 6(c+d)^2$  is required in the binomial expansion of the root, namely  $[(a+b) + (c+d)]^4$ . As part of computing the b-list  $6(c+d)^2$  directly, we substitute  $c \cdot 12d$  for  $c \cdot 2d$  in the pgt for  $c + d$ , and so on. We defer all binomial coefficient multiplications associated with retagging columns, e.g.,  $6(c^2 + d^2)$ , to Step 5.

In consequence, the retagging which normally follows the processing of all product columns of all nodes at a level will have to be modified somewhat so as to be able to retrieve, say,  $c^2$  and  $d^2$  separately from  $12\ cd$ . At the end of Step 4, we have computed the a-lists and some terms of the b-lists to be used at level 0.

#### Step 5 - Processing of the Root Node

As before, we write

$$f^n = f_1^n + f_2^n + \sum_{i=1}^{n-1} a\text{-list}_i \cdot b\text{-list}_i.$$

This is the usual binomial expansion in which the binomial coefficients have been included in the b-lists. The polynomials  $f_1^n, f_2^n$ , and all a-lists have been computed. We use a single constant-product N-split to perform all the binomial coefficient multiplications necessary to complete the computation of the b-lists. We load the first two b-lists into the term buffer. We concatenate the first two a-lists to form a long polynomial. We use a single list-product N-split to compute the first two list products; the answers may be written to disc. We continue in this way, two-by-two, until the  $n-1$  list products have been computed. This completes the processing of the root.

### 9.3. ANALYSIS OF THE SPEED-UP RATIO

We can obtain the speed-up ratio for Associative F by dividing the number of elementary operations in the sequential algorithm (BINF) by the number of firings of the PPA in the parallel algorithm. We denote these two quantities by  $F(t,n)$  and  $P(t,n)$ , respectively. We assess the speed-up ratio  $F(t,n)/P(t,n)$  by first calculating and then approximating the function  $P(t,n)$ . There are  $n-1$  firings of the PPA to process the terminal nodes. We calculate the number of firings required to process a node, at any level from  $k-1$  to 2, when the sub-node size is  $s$ . We calculate first the number of firings for all binomial work in a pgt. This is given by

$$BW(s) = \sum_{i=1}^{n-1} \left\lceil \frac{1}{N} \cdot (n-i) \cdot \binom{s+u-1}{u} \right\rceil, \text{ where } u = \lceil i/2 \rceil \quad (9.3.1)$$

This is just the sum over product columns of ceiling of  $1/N$  of the size of all b-lists in that column. Next, we calculate the number of firings for all non-binomial work. This is given by

$$NBW(s) = \sum_{i=1}^{n-1} \binom{s+u-1}{u} \cdot \left\lceil \frac{1}{N} \sum_{j=v}^{n-u} \binom{s+j-1}{j} \right\rceil, \text{ where } u = \lceil i/2 \rceil \quad (9.3.2)$$

and  $v = \lceil (i+1)/2 \rceil$ . This is just the sum over product columns of the size of the b-list times ceiling of  $1/N$  of the size of all a-lists in that column. We now have, for each node, the number of firings in all constant-product  $N$ -splits, and the number in all list-product  $N$ -splits.

The two formulas BW(s) and NBW(s) need to be summed over all interior nodes from level k-1 to level 2 for the total number of firings of the PPA in the classical pgt work. Since the processing at level 1 differs only in the substitution of distribution coefficients, the sum may be extended to level 1. The number of firings in the single constant-product N-split at the root level is given by

$$\left\lceil \frac{1}{N} \cdot [4 \cdot \text{group}(t/4, r-1) + 2N_n \cdot \text{size}(t/4, r)] \right\rceil \quad (9.3.3)$$

where  $r = \lceil n/2 \rceil$  and  $N_n = 1(0)$  when  $n$  is even (odd). This is just ceiling of  $1/N$  times all root binomial work in either of the F algorithms (BINF or Associative F). The number of firings in the r-1 or r list-product N-splits at the root level is given by

$$\sum_{j=1}^{r-1} \binom{t/2+j-1}{j} \cdot \left\lceil \frac{1}{N} \cdot 2 \binom{t/2+n-j-1}{n-j} \right\rceil + N_n \cdot \binom{t/2+r-1}{r} \cdot \left\lceil \frac{1}{N} \binom{t/2+r-1}{r} \right\rceil \quad (9.3.4)$$

where  $r$  and  $N_n$  have the same meanings as above. If we perform the indicated summation of BW(s) and NBW(s), and add the firings for the root node and the terminal nodes, we obtain finally an expression for  $P(t, n)$ . This is given by

$$P(t,n) = \sum_{j=1}^{r-1} \binom{t/2+j-1}{j} \cdot \left\lceil \frac{2}{N} \binom{t/2+n-j-1}{n-j} \right\rceil + N_n \cdot \binom{t/2+r-1}{r}.$$

$$\cdot \left\lceil \frac{1}{N} \binom{t/2+r-1}{r} \right\rceil + \left\lceil \frac{1}{N} \cdot [4 \cdot \text{group}(t/4, r-1) + \right.$$

$$+ 2N_n \cdot \text{size}(t/4, r)] \right\rceil + \sum_{j=1}^{k-1} 2^j [\text{BW}(t/2^{j+1}) +$$

$$+ \text{NBW}(t/2^{j+1})] + n - 1 \quad (9.3.5)$$

As before, we obtain a lower bound on the speed-up ratio  $F(t,n)/P(t,n)$  by obtaining an upper bound on the quantity  $P(t,n)$ . In general,  $\lceil s/p \rceil < s/p + 1$ . We make systematic substitutions of this inequality for each occurrence of the ceiling function to obtain an upper bound of the form  $P(t,n) < F(t,n)/N + Q(t,n)$ . The contribution to  $Q(t,n)$  from the root is just  $\text{group}(t/2, r-1) + N_n \cdot \text{group}(t/2, r) + 1$ . As explained in the multiprocessor analysis, the terminal nodes give rise to a contribution of  $(N-t)(n-1)/N$ . If we refer back to the form of  $\text{BW}(s)$  and  $\text{NBW}(s)$ , we see that each interior node makes a contribution of  $n-1 + 2 \cdot \text{group}(s, r-1) + N_n \cdot \text{size}(s, r)$ , where  $s$ , as always, is the sub-node size. The term  $n-1$  comes from substituting the ceiling inequality in  $\text{BW}(s)$ , the remaining terms from substituting the same inequality in  $\text{NBW}(s)$ .

Therefore, we may write

$$\begin{aligned}
 Q(t,n) &= \text{group}(t/2, r-1) + N_n \cdot \text{size}(t/2, r) + 1 \\
 &+ \sum_{j=1}^{k-1} 2^j [n-1 + 2 \cdot \text{group}(t/2^{j+1}, r-1) + \\
 &+ N_n \cdot \text{size}(t/2^{j+1}, r)] + (N-t)(n-1)/N \quad (9.3.6)
 \end{aligned}$$

The summation may be written as

$$\begin{aligned}
 (n-1)(t-2) + \frac{t}{2} \cdot \sum_{m=0}^{k-2} 2^{-m} [2 \cdot \text{group}(2^m, r-1) + \\
 + N_n \cdot \text{size}(2^m, r)] \quad (9.3.7)
 \end{aligned}$$

Using the closed forms of  $\text{group}(s, n)$  and  $\text{size}(s, n)$  this is

$$(n-1)(t-2) + \frac{t}{2} \cdot \sum_{m=0}^{k-2} \sum_{j=1}^r b_j^{(r)} (2^m)^{j-1} \quad (9.3.8)$$

where

$$b_j^{(r)} = \frac{2}{(r-1)!} \left[ \begin{matrix} r \\ j+1 \end{matrix} \right] + \frac{N_n}{r!} \left[ \begin{matrix} r \\ j \end{matrix} \right], \quad \text{and} \quad \left[ \begin{matrix} r \\ r+1 \end{matrix} \right] = 0.$$

Simplifying gives

$$\begin{aligned}
 (n-1)(t-2) + \frac{t}{2} \cdot b_1^{(r)} \cdot k + \frac{t}{2} \cdot \sum_{j=1}^{r-1} b_{j+1}^{(r)} \cdot \frac{(t/2)^{j-1}}{2^{j-1}} \quad (9.3.9)
 \end{aligned}$$

Finally then

$$\begin{aligned}
Q(t,n) = & \text{group}(t/2, r-1) + N_n \cdot \text{size}(t/2, r) + \\
& + (N-t)(n-1)/N + \frac{t}{2} \cdot \sum_{j=1}^{r-1} b_{j+1}^{(r)} \cdot \frac{(t/2)^{j-1}}{2^{j-1}} + \\
& + (n-1)(t-2) + 1 + \frac{t}{2} \cdot b_1^{(r)} \cdot k
\end{aligned} \tag{9.3.10}$$

Since  $P(t,n) < F(t,n)/N + Q(t,n)$ , we have the following lower bound on the speed-up ratio for Associative F.

$$\frac{F(t,n)}{P(t,n)} > N \cdot \frac{F(t,n)}{F(t,n) + N \cdot Q(t,n)} \tag{9.3.11}$$

This speed-up ratio is, clearly, less attractive than the speed-up ratio for Multiprocessor E. The leading term of  $Q(t,n)$  is given by

$$\frac{1}{P!} \left(\frac{t}{2}\right)^P \cdot \left[1 + \frac{2-N_n}{2^{P-1}-1}\right], \text{ where } P = \lfloor n/2 \rfloor \tag{9.3.12}$$

In comparison, the leading term of  $F(t,n)$  is  $t^n/n!$ . Therefore, for large  $t$  and large  $n$ ,  $Q(t,n)$  will be negligible in comparison with  $F(t,n)$ . It follows that, asymptotically, the speed-up ratio for Associative F approaches the theoretical ideal. The quantity  $N \cdot Q(t,n)$  is vastly larger than  $2kN + (N-t)(n-1)$ ; the corresponding quantity in the formula for the lower bound on the speed-up ratio for Multiprocessor E. It cannot automatically be concluded that the multiprocessor parallel architecture is superior to the associative-processor architecture for this computation; one would need to make a careful study

of the hidden overhead in Multiprocessor E. It seems that the overhead in Associative F is considerably less. This may ultimately tip the scales in favour of Associative F.

Several modifications and improvements of algorithm Associative F suggest themselves. One could adopt a unified strategy by processing pairs of product columns in the pgt; this is the way list products are computed at the root level in the present version. The pairwise strategy would reduce the contribution of each node to  $Q(t,n)$  to  $n-1 + \text{group}(s,r-1) + N_n \cdot \text{size}(s,r)$ , which affects the coefficient of the leading term of  $Q(t,n)$ . (In (9.3.12),  $2-N_n$  becomes 1). In case the term buffer is not large enough, a simple modification of the architecture, namely, providing separate match indicators for a-lists and b-lists, allows one to segment the subcomputations as they are currently defined, and have piecemeal loading of parts of a-lists and parts of b-lists, alternately, without additional associative searching. Finally, in choosing between Multiprocessor E and Associative F, one must consider the machine cost. The processing elements of the PPA are simpler than the slave processors of the multiprocessor machine. Thus, it is entirely possible that, from an economic standpoint, one could achieve much larger multiplicity with the associative architecture. We believe that Associative F is a strong argument in favour of the very great suitability of an associative parallel architecture for this class of computations, more suitable, in fact, than the multiprocessor architecture.



We have been careful not to claim that algorithm Associative F is the optimal way to exploit the proposed special-purpose associative-processor architecture. Yet clearly it is a good way, for the following reason. We make best use of the PPA by minimizing the number of times it is fired when it is less than completely full. Associative F adopts the strategy of processing the list products in a pgt at most two product columns at a time. This ensures that the b-lists in the set of list products are all of precisely one size, say,  $b$ . It will require some number of loads of the  $t_1$  fields of the PPA to exhaust the a-lists in the set of list products. For each such load, except possibly the last, there will be  $b$  firings of the PPA which use its capacity to the fullest. Moreover, the initializations of the  $t_2$  fields of the PPA prior to each firing are straightforward: Let  $N_1$   $t_1$ -fields of the PPA contain all or part of an a-list,  $a_i$ ; the  $N_1$  matching  $t_2$ -fields are all filled with the next term of the corresponding b-list,  $b_i$ . With trivial random-access initialization of the  $t_2$  fields, we get the benefit of  $b$  firings of all  $N$  cells of the PPA before the next load from the AM. We have reason to believe, then, that we are making good use of this parallel machine, and that processing two product columns at a time is the best we can do.

The space requirements for Associative F are not absolute, in the following sense. Normally, to process a set of list products by the method outlined above, one needs a term buffer able to hold all the b-lists belonging to the set,

and all at once. When a buffer of this size is not available, the computation may be performed in a piecemeal fashion, with a resultant decrease in speed-up ratio. The discussions relative to the space complexity of the sequential algorithm BINP show that one could not hope to implement the parallel algorithm Associative F, as it now stands, with less than  $\text{size}(t/2, p) + 4.\text{group}(t/4, n-1)$  cells of associative memory, nor less than  $\text{size}(t/2, p)$  cells of term buffer, where  $p = \lfloor n/2 \rfloor$ . The two requirements are additive. Algorithm modification amounting to space-time trade-off has already been mentioned; given the lower cost of buffer cells, it is probably not a good idea to make up for insufficient buffer size by increasing the size of the associative memory. A somewhat generous estimate for the space complexity of Associative F, then, if we combine the two forms of memory, is given by

$$2.\text{size}(t/2, p) + 4.\text{group}(t/4, n-1), p = \lfloor n/2 \rfloor$$

(9.3.13)

CHAPTER X

CONCLUSION

## CHAPTER X

## CONCLUSION

We have restricted ourselves to the problem of the symbolic computation of integer powers of completely or almost completely sparse multivariate polynomials. Six new algorithms for this problem, namely, the four sequential algorithms BINC, BIND, BINE, and BINF, and the two parallel algorithms Multiprocessor E and Associative F, have been proposed and investigated. The two parallel algorithms are specifically intended to be run on two special-purpose parallel machines, also discussed in this thesis. All six algorithms are based on the idea of using binomial expansion as the fundamental and exclusive tool for computing powers of polynomials, for the desired power of the original polynomial, for powers of subpolynomials (other than monomials) arising in the original binomial expansion, and so on, recursively. Previous analysis did suggest the superiority of binomial expansion as a general approach. However, the previous best sequential binomial-expansion algorithm, namely, BINB, did not carry the binomial-expansion approach through systematically, i.e., recursively, and so did not realize the full benefits of the binomial-expansion approach. The four new sequential algorithms are successive refinements and improvements of the systematic binomial-expansion approach.

The main conclusions for the sequential algorithms may be summarized as follows. Both the time complexity and the

space complexity of the algorithms depend on the design decisions relating to polynomial splitting, subpolynomial powering, and cross-product formation, as discussed above. When the polynomials are sparse, even splitting is to be preferred because of the substantial reduction in the cost of powering subpolynomials; this idea is used in all of these algorithms, including BINB. Repeated multiplication is not a good way to obtain powers of subpolynomials, both because of the excessive number of coefficient multiplications required, and because of the need to sort the intermediate results; this less than optimal approach is used in BINB, but in none of the new algorithms. BINC and BIND both use recursion to generate powers of subpolynomials. BINE uses a modified form of recursion akin to dynamic programming. BINF uses a modified form of dynamic programming which avoids some intermediate steps. In this series, the time complexity decreases each time. We think it is very unlikely that there is something better than dynamic programming for computing the powers of subpolynomials, given the computational and cost models which have been adopted in the thesis, and discussed above.

BINC, alone among the new algorithms in this respect, does not use the rather obvious improvement of always multiplying the binomial coefficient by the smaller polynomial first; BINB also fails to take advantage of this improvement. When we analyse the time complexities of the five sequential binomial-expansion algorithms BINB, BINC, BIND, BINE, and BINF, we

find that the respective cost functions form a strictly monotonically decreasing (finite) sequence. The new algorithms have the further advantage that, when the polynomials are sparse, they do not, unlike BINB, require exponent comparisons. If one is generous in assigning a low space complexity to BINB, then only sophisticated implementations of BINE, and the standard implementation of BINF, have lower space complexities. BINF does very well from a space-complexity standpoint; the modified form of dynamic programming it uses was specifically introduced to save space, not time. We conjecture that BINF is optimal for both time and space among sequential binomial-expansion algorithms. The cost comparisons have been made using both comparison of leading terms of analytically-obtained cost functions, and straight tabulation; the two methods do not give different results.

In the parallel case, the aim is to devise parallel algorithms in conjunction with special-purpose parallel architectures; parallel solutions to the problem of powering sparse polynomials appear not to have been studied elsewhere. The two parallel algorithms Multiprocessor E and Associative F both have speed-up ratios which, asymptotically, approach the theoretical upper limit. Although the calculated speed-up ratio of the former seems to suggest that Multiprocessor E is the superior of the two algorithms, we believe that the time complexity hidden in the Multiprocessor E overhead, and hardware costs, both shift the balance in favour of Associative F. The space complexities of the two parallel algorithms, apparently,

are not enormously different from the space complexities of their sequential counterparts. The main parallel result is, not to exhibit optimal parallel algorithms, but rather to show convincingly how very well-suited these or similar parallel algorithms, running on the two proposed parallel machines, are to the general problem of powering sparse polynomials. It seems also that good sequential algorithms for this problem may be adapted, more or less directly, to yield good parallel algorithms. The whole parallel area is wide open for further research.

This thesis has been intended as a contribution to theoretical computational complexity. It is the feeling among computer scientists today that no new algorithm can be respectably presented without some analysis of its behaviour. While not rejecting the approach which is based on empirical comparisons of (hopefully reasonable) implementations of competing algorithms, our approach to algorithm analysis has been essentially to obtain the analytically-exact cost functions, and compare them. However, we have tested the performance of algorithm BINE with a full implementation in PASCAL 6000. We state again that major portions of this thesis have previously been published in [2]; a short comparison of the earlier and later write-ups occurs at the end of Chapter 1.

REFERENCES



REFERENCES<sup>1</sup>

- [1] Aho, A., Hopcroft, J., and Ullman, J., The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974, pp.311-313.
- [2] Alagar, V., and Probst, D., "Binomial-Expansion Algorithms for Computing Integer Powers of Sparse Polynomials", in International Computing Symposium 1977, E. Morlet and D. Ribbens, eds., North-Holland, Amsterdam, 1977, pp.395-402.
- [3] Dijkstra, E.W., "Structured Programming", in Software Engineering Techniques, J.N. Buxton and B. Randell, eds., NATO Science Committee, Brussels, 1970, pp.84-88.
- [4] Fateman, R.J., "On the Computation of Powers of Sparse Polynomials", Studies in Appl.Math., Vol.53, No. 2, June 1974, pp.145-155.
- [5] Fateman, R.J., "Polynomial Multiplication, Powers and Asymptotic Analysis: Some Comments", SIAM J. Comput., Vol.3, No. 3, Sept. 1974, pp.196-213.
- [6] Flynn, M.J., "Some Computer Organizations and Their Effectiveness", IEEE Trans. on Comput., Vol. C-21, No. 9, Sept. 1972, pp.948-960.
- [7] Gentleman, W.M., "Optimal Multiplication Chains for Computing a Power of a Symbolic Polynomial", Math. of Comput., Vol.26, No.120, Oct.1972, pp.935-939.
- [8] Gentleman, W.M., "On the Relevance of Various Cost Models of Complexity", in Complexity of Sequential and Parallel Numerical Algorithms, J.F. Traub, ed., Academic Press, New York, 1973, pp.103-109.
- [9] Heindel, L.E., "Computation of Powers of Multivariate Polynomials Over the Integers", J.Comp.Syst.Sci., Vol. 6, 1971, pp.1-8.

---

<sup>1</sup>References [8],[9],[11], and [15] have not been cited in the text.

- [10] Horowitz, E., and Sahni, S., "The Computation of Powers of Symbolic Polynomials", SIAM J. Comput., Vol. 4, No. 2, June 1975, pp.201-208.
- [11] Katz, J.H., "Matrix Computations on an Associative Processor", in Parallel Processor Systems, Technologies, and Applications, L.C. Hobbs, et al., eds., Spartan Books, New York, 1970, pp.131-149.
- [12] Knuth, D.E., The Art of Computer Programming, Vol.1, Fundamental Algorithms, 2nd edition, Addison-Wesley, Reading, Mass., 1975, pp.65-67.
- [13] Parnas, D.L., "On the Design and Development of Program Families", IEEE Trans. on Soft.Eng., Vol. SE-2, No.1, Jan. 1976, pp.1-9.
- [14] Stone, H.S., "Problems of Parallel Computation", in Complexity of Sequential and Parallel Numerical Algorithms, op.cit., pp.1-15.
- [15] Thurber, K.J., and Wald, L.D., "Associative and Parallel Processors", Comput.Surv., Vol.7, No.4, Dec.1975, pp.215-255.

APPENDIX I

VALUES OF  $B(t,n)$ ,  $C(t,n)$ ,  $D(t,n)$ ,  $E(t,n)$ , AND  
 $L(t,n)$  FOR SELECTED VALUES OF  $t$  AND  $n$ .

O	T	N	B	C	D	E	L
4	4	69	98	96	68	31	
8	4	550	608	586	458	322	
16	4	5878	5116	4944	4400	3860	
17	4	7334	6252	6065	5434	4828	
18	4	8980	7613	7366	6648	5967	
19	4	10961	9150	8865	8060	7296	
20	4	13173	11070	10584	9692	8835	
21	4	15792	13058	12544	11555	10605	
22	4	18687	15357	14768	13682	12628	
23	4	22068	17922	17277	16494	14927	
24	4	25774	21604	20098	18818	17526	
25	4	30052	24696	23265	21865	20450	
26	4	34708	28193	26796	25276	23725	
27	4	40029	32054	30715	29075	27378	
28	4	45785	36494	35054	33294	31437	
29	4	52306	41206	39839	37947	35931	
30	4	59323	46441	45104	43080	40890	
31	4	67212	52162	50876	48720	46345	
32	4	75662	58424	57192	54904	52328	
4	5	110	178	174	110	52	
8	5	1254	1419	1364	1036	784	
16	5	21870	18988	18412	16852	15488	
17	5	28687	24403	23796	21948	20332	
18	5	36664	31172	30320	28184	26316	
19	5	46830	39282	38247	35823	33630	
20	5	58558	49639	47714	45002	42484	
21	5	73178	61715	59076	56034	53109	
22	5	89846	75003	72462	69090	65758	
23	5	110245	91200	88247	84545	80707	
24	5	133270	114527	106632	102600	98256	
25	5	161013	135572	128144	123688	118730	
26	5	192060	160107	152932	148052	142480	
27	5	228972	188307	181503	176199	169884	
28	5	269974	221274	214134	208406	201348	
29	5	318160	258144	251457	245243	237307	
30	5	371338	300242	293740	287040	278226	
31	5	433203	347865	341642	334456	324601	
32	5	501086	401652	395528	387856	376960	
4	6	162	290	284	164	80	
8	6	2574	2923	2820	2140	1708	
16	6	71462	62538	61084	57508	54248	
17	6	98238	84484	82942	78617	74596	
18	6	131020	113115	110830	105756	100929	
19	6	174670	149296	146296	140473	134577	
20	6	227198	196715	190672	184100	177080	
21	6	295333	253107	246115	238660	230209	
22	6	376143	323243	314274	305936	295988	
23	6	478702	409016	397801	388580	376717	
24	6	598838	532548	498956	488852	474996	
25	6	748521	652920	621356	610067	593750	

T	N	B	C	D	E	L
26	6	921984	797703	767598	755124	736255
27	6	1134722	970146	941871	928212	906165
28	6	1378956	1176437	1147714	1132870	1107540
29	6	1674418	1417035	1390442	1374181	1344875
30	6	2010833	1699647	1674210	1656532	1623130
31	6	2412970	2029322	2005249	1986154	1947761
32	6	2867502	2412504	2388784	2368272	2324752
4	7	226	439	430	230	116
8	7	4894	5529	5344	4104	3424
16	7	211774	188160	184740	177596	170528
17	7	304397	266707	263151	254333	245140
18	7	423376	373591	367898	357404	346086
19	7	587993	515312	507323	495157	480681
20	7	795138	706573	689254	675414	657780
21	7	1073406	947303	926066	910192	888009
22	7	1417480	1257875	1228658	1210750	1184018
23	7	1868465	1652876	1613841	1593899	1560757
24	7	2417758	2223285	2097304	2075328	2035776
25	7	3122905	2820994	2702731	2677906	2629550
26	7	3970588	3563212	3450664	3422990	3365830
27	7	5039559	4476529	4371619	4341096	4272021
28	7	6309938	5598258	5493950	5460578	5379588
29	7	7887388	6953662	6858651	6821288	6724491
30	7	9743128	8591384	8500808	8460254	8347650
31	7	12016489	10557437	10472904	10428759	10295441
32	7	14666878	12906000	12821992	12774256	12620224

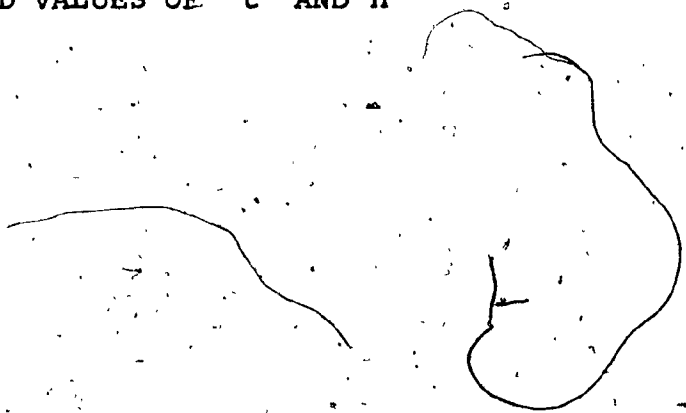
APPENDIX II

VALUES OF  $E(t,n) - F(t,n)$  FOR SELECTED  
VALUES OF  $t$  AND  $n$

T	N	$E(T,N)-F(T,N)$
16	5	33
17	5	37
18	5	41
19	5	46
20	5	51
21	5	56
22	5	61
23	5	67
24	5	73
25	5	79
26	5	85
27	5	92
28	5	99
29	5	106
30	5	113
31	5	121
32	5	129
16	6	114
17	6	118
18	6	167
19	6	157
20	6	202
21	6	207
22	6	278
23	6	263
24	6	326
25	6	332
26	6	429
27	6	408
28	6	492
29	6	499
30	6	626
31	6	598
32	6	706
16	7	194
17	7	228
18	7	262
19	7	307
20	7	352
21	7	402
22	7	452
23	7	515
24	7	578
25	7	647
26	7	716
27	7	800
28	7	884
29	7	975
30	7	1066
31	7	1174
32	7	1282

APPENDIX III

VALUES OF  $S_B(t,n)$ ,  $S_E(t,n)$ ,  $\underline{S}_E(t,n)$ , AND  
 $S_F(t,n)$  FOR SELECTED VALUES OF  $t$  AND  $n$





1

ABSTRACT

SEQUENTIAL AND PARALLEL ALGORITHMS FOR SYMBOLIC  
COMPUTATION OF INTEGER POWERS OF SPARSE POLYNOMIALS

David Karl Probst

This thesis proposes four new sequential algorithms, and two new parallel algorithms, for symbolic computation of integer powers of completely or almost completely sparse polynomials in one or several variables, and analyses their time complexity and space complexity. The two parallel algorithms are specifically intended to be run on two proposed special-purpose parallel machines, also described in the thesis. We conjecture that one of the new sequential algorithms is optimal for time and space within the family of algorithms which adopt a binomial-expansion approach to computing integer powers of sparse polynomials. If the original sparse polynomial consists of  $t$  monomials, then the time complexity of computing the  $n^{\text{th}}$  power using the best new sequential algorithm is given by:

$$\frac{t^n}{n!} + t^{n-1} \left[ \frac{1}{2(n-2)!} + \frac{1}{2^{n-2}(n-1)!} \right] + O(t^{n-2}), \text{ for } n > 2$$

The space complexity for the same task and the same algorithm is given by:

$$\frac{t^{n-1}}{2^{2n-4}(n-1)!} + O(t^{n-2}), \text{ for } n > 2$$

The two new parallel algorithms, one for a multiprocessor machine and one for an associative-processor machine, both have speed-up ratios which asymptotically approach the theoretical ideal of a speed-up ratio of  $N$  for  $N$  processors. The aim here has been, not to exhibit optimal algorithms, but merely to demonstrate the extreme suitability of a parallel approach to this problem. The space complexities of the parallel algorithms do not greatly exceed the space complexities of their sequential counterparts. We give arguments which suggest that the associative-processor architecture may be the preferred architecture of the two.

ACKNOWLEDGEMENTS

## ACKNOWLEDGEMENTS

I gratefully acknowledge the invaluable assistance of my thesis adviser, Prof. V.S. Alagar, in all phases of the preparation of this thesis. In particular, I acknowledge Dr. Alagar's suggestion of the problem, and constant collaboration thereafter. This collaboration took the form of the two of us proposing ideas, and constantly discussing and re-discussing them. Nearly all of the results which have been obtained here are the fruit of these discussions. I also acknowledge considerable financial assistance from Dr. Alagar.

In addition, I should like to acknowledge the contribution of the Department Chairman, Prof. H.S. Heaps, in two respects: (1) initial financial support, and (2) Prof. Heaps' policy of actively encouraging research by students in the Department.

TABLE OF CONTENTS

## TABLE OF CONTENTS

	PAGE
ABSTRACT . . . . .	i
ACKNOWLEDGEMENTS . . . . .	iii
LIST OF FIGURES . . . . .	vi
LIST OF TABLES . . . . .	vii
NOTATIONS . . . . .	viii
CHAPTER I INTRODUCTION . . . . .	1
CHAPTER II NOTATION AND ELEMENTARY RESULTS . . . . .	11
CHAPTER III STATE OF THE PROBLEM PRIOR TO THE PRESENT WORK . . . . .	19
3.1 Algorithm RMUL . . . . .	20
3.2 Algorithm RSQ . . . . .	20
3.3 Algorithm NOMA . . . . .	22
3.4 Algorithm NOMB . . . . .	22
3.5 Algorithm BINA . . . . .	23
3.6 Algorithm BINB . . . . .	24
CHAPTER IV THE ALGORITHM FAMILY . . . . .	26
4.1 Design Decisions . . . . .	29
4.1.1 General Approach . . . . .	29
4.1.2 Splitting . . . . .	30
4.1.3 Powers of Subpolynomials . . . . .	34
4.1.4 Combining Powers . . . . .	37
4.1.5 Optimum Design Decisions . . . . .	39
CHAPTER V TIME COMPLEXITY OF ALGORITHMS (SEQUENTIAL OPTION) . . . . .	41
5.1 Algorithm BINB . . . . .	41
5.2 Algorithm BINC . . . . .	43
5.3 Algorithm BINS . . . . .	48
5.4 Algorithm BIND . . . . .	49
5.5 Algorithm BINE . . . . .	54
5.6 Algorithm BINF . . . . .	66

	PAGE
CHAPTER VI SPACE COMPLEXITY OF ALGORITHMS (SEQUENT- IAL OPTION) . . . . .	73
6.1 Space Analysis for BIN <sub>A</sub> and BIN <sub>B</sub> . . . . .	78
6.2 Space Analysis for BIN <sub>E</sub> and BIN <sub>F</sub> . . . . .	82
CHAPTER VII PARALLEL ALGORITHMS . . . . .	95
CHAPTER VIII MULTIPROCESSOR OPTION (MULTIPROCESSOR E). . . . .	100
8.1 Description of the Architecture . . . . .	100
8.2 Description of the Algorithm . . . . .	105
8.3 Analysis of the Speed-Up Ratio . . . . .	108
CHAPTER IX ASSOCIATIVE-PROCESSOR OPTION (ASSOCIATIVE F) . . . . .	114
9.1 Description of the Architecture . . . . .	114
9.2 Description of the Algorithm . . . . .	120
9.3 Analysis of the Speed-Up Ratio . . . . .	125
CHAPTER X CONCLUSION . . . . .	133
REFERENCES . . . . .	137
APPENDIX I VALUES OF $B(t,n)$ , $C(t,n)$ , $D(t,n)$ , $E(t,n)$ , AND $L(t,n)$ FOR SELECTED VALUES OF $t$ AND $n$ . . . . .	139
APPENDIX II VALUES OF $E(t,n) - F(t,n)$ FOR SELECTED VALUES OF $t$ AND $n$ . . . . .	141
APPENDIX III VALUES OF $S_B(t,n)$ , $S_D(t,n)$ , $S_E(t,n)$ , AND $S_F(t,n)$ FOR SELECTED VALUES OF $t$ AND $n$ . . . . .	142

LIST OF FIGURES



## LIST OF FIGURES

NUMBER		PAGE
4.1	Three term-group trees . . . . .	35
4.2	The program family tree . . . . .	39
5.1	The power group triangle when $n = 6$ . . .	57
5.2	A term-group tree . . . . .	59
5.3	A term-group tree . . . . .	62
8.1	System diagram . . . . .	101
8.2	Power group triangle (pgt) for $n = 4$ . .	104
9.1	System diagram . . . . .	115
9.2	Pgt for $n = 5$ . . . . .	117

LIST OF TABLES

## LIST OF TABLES

NUMBER		PAGE
2.1	Values of a few Stirling numbers of the first kind . . . . .	13
5.1	Itemizing the costs in BINB . . . . .	42
5.2	Itemizing the costs in BINC . . . . .	44
5.3	Leading terms for $B(t,n), C(t,n), L(t,n)$ ;	47
5.4	Itemizing the costs in BIND . . . . .	50

NOTATIONS

## NOTATIONS

f	Original polynomial
g	Arbitrary polynomial
k	Logarithm of t (base 2) <u>or</u> dummy variable
n	The power sought
p	$\lfloor n/2 \rfloor$ <u>or</u> dummy variable
r	$\lfloor n/2 \rfloor$ <u>or</u> dummy variable
s	Subnode size <u>or</u> dummy variable
t	Number of monomials in f
a-list, b-list	In $\binom{n}{r} f_1^r f_2^{n-r}$ , if $\text{size}(f_1^r) > \text{size}(f_2^{n-r})$ , then $f_1^r$ is the a-list, and $\binom{n}{r} f_2^{n-r}$ is the b-list. Ties are broken by the form of the pgt
group(s,n)	$\sum_{j=1}^n \text{size}(s,j)$
pgt	Power group triangle
size(f)	Number of monomials in the polynomial f
size(s,n)	An alternate functional form. If $\text{size}(f) = s$ , this is synonymous with $\text{size}(f^n)$
t <sub>1</sub> fields, t <sub>2</sub> fields	Registers in the PPA
AM	Associative memory
BB	Binomial-binomial work
BC(s,n)	Binomial coefficient work to process a node whose subnodes are of size s
BINA, ..., BINF	Sequential algorithm names
BW	Binomial work

$B(t,n), \dots, F(t,n)$	Sequential algorithm time complexities
CU	Control unit
E	Number of CM words for exponent set
GBW	Group binomial work
$L(t,n)$	Theoretical lower limit for sequential algorithm time complexity
M	$N/2^j$ (Multiprocessor architecture only)
MIMD	Multiple instruction stream, multiple data stream
$M_n$	Equals 1(3) when n is even(odd)
Multiprocessor Associative	Parallel algorithm names
N	Parallel processor multiplicity. (Multiprocessor and associative-processor architectures)
NBW	Nonbinomial work
$N_n$	Equals 1(0) when n is even(odd)
N-split	Dividing an amount of work among N processors. There are multiprocessor N-splits and PPA N-splits
P	Number of CM words for link field
PPA	Parallel processing array
$P(t,n), T(t,n)$	Parallel algorithm time complexities
RBW	Root binomial work
$S_A, \dots, S_F$	Sequential algorithm space complexities
SIMD	Single instruction stream, multiple data stream

x

$\delta_p$	Equals 0(1) when p is even (odd)
$\sigma_j^{(n)}$	The $j^{\text{th}}$ symmetric function over n
$\binom{n}{r}$	Binomial coefficient n choose r
$[n]_r$	Stirling number of the first kind
$\{n\}_r$	Stirling number of the second kind
$O(\dots)$	Big O notation of Bachmann
$\lfloor \dots \rfloor$	Floor of ...
$\lceil \dots \rceil$	Ceiling of ...

To my wife and our  
forthcoming child



CHAPTER I

INTRODUCTION

## CHAPTER I INTRODUCTION

This thesis contains results arising from the investigation and analysis of new algorithms for computing powers of symbolic polynomials on both sequential and parallel machines; it is intended as a contribution to theoretical computational complexity which would not be irrelevant to practical problems of algorithm assessment and programming. Previous attempts to determine best algorithms for polynomial powering on sequential machines have been motivated by a desire to obtain system programs for use in symbolic algebraic manipulation systems. Parallel polynomial powering algorithms have been much less studied, if at all; the aim here is to devise more or less special-purpose computer architectures, and then formulate parallel algorithms especially suited to them. It is noteworthy that efficient serial algorithms are not necessarily extendable to obtain efficient parallel algorithms. The parallel algorithms presented here are obtained by finding or creating parallelism in the sequential algorithms, and then exploiting it.

The choice of an appropriate computer algorithm will determine the size and complexity of problems which can be solved in a reasonable time. The approach taken in this thesis is to choose one out of a set of competing algorithms on the basis of a theoretical analysis of the intrinsic algorithm

2

complexity rather than, say, on the basis of run-time tests. In order that the problems of algorithm analysis be well-posed, it is necessary to specify both a computational model, which characterizes the problem domain, and a cost model, which provides the criterion or criteria by which intrinsic difficulty is to be measured. The computational model is that the input polynomials be completely sparse polynomials (definition follows) in one or more variables (indeterminates) with integer coefficients. The cost model is that the algorithms will be analysed in terms of the number of coefficient multiplications required to compute the final result - in the parallel case, in terms of the number of parallel cycles needed to perform these multiplications. Both models will now be explained.

Completely dense and completely sparse polynomials are the two basic complementary computational models for which one does an analysis of powering algorithms. Sparse polynomials have few non-zero coefficients; completely dense polynomials have no zero coefficients. A univariate (one-variable) polynomial of degree  $d$  is completely dense if it has no zero coefficients. That is, it has  $d + 1$  terms. A multivariate polynomial with  $v$  variables where each variable  $x_1, \dots, x_v$  occurs to maximum degree  $d$  will be completely dense if all possible terms are present. In that case,

$$f = f_d x_v^d + \dots + f_0,$$

where the  $f_i$  are completely dense in  $v-1$  variables, and

$$\text{size}(f) = (d+1)^v,$$

where  $\text{size}(f)$  is the number of monomial terms in  $f$ . If  $f^i$  has  $(d+1)^v$  terms for  $1 \leq i \leq n$ , then  $f^i$  remains completely dense to power  $n$ . This is the worst case assumption of polynomial growth. Dense polynomials in 3 or more variables can only be raised to quite small powers before either core or time become excessive. [10]

In sparse polynomials, whatever the number of variables (indeterminates) in the polynomial, and whatever the degree of the polynomial in each variable (indeterminate), there are only  $t$  non-zero terms. We represent a polynomial in the class of sparse polynomials as a sum of monomials, where each monomial is of the form  $c(\prod x_i^{\alpha_i})$ ,  $\alpha_i \geq 0$ ,  $c$  and  $\alpha_i$  integers. A sparse polynomial may be characterized by the number of non-zero terms,  $t$ , in its representation as a sum of monomials. We say that a polynomial  $f$  is completely sparse to power  $n$  if  $f^i$ , fully expanded, for all  $i$ ,  $1 \leq i \leq n$ , contains exactly the number of terms of the  $t$ -term multinomial expansion. To say that the number of terms is given by the size of the  $t$ -term multinomial expansion is to say that no further collection of like terms is possible. Multivariate problems dealt with by symbolic algebraic manipulation systems are often sparse in character. The theoretical model of 'sparse polynomial' is due to Morven Gentleman. [7]

4

The assumption in the computational model of complete sparsity of the input polynomials affects the design and analysis of the powering algorithms in two ways. From the standpoint of design, sparsity guarantees that no like terms will ever be formed through multiplying one subpolynomial by another, given the particular structure of the new binomial-expansion algorithms. This will be explained below. From the standpoint of analysis, sparsity allows us to predict the sizes of all intermediate results, and thus carry through the analysis in a mathematically tractable way. If the input polynomial is more or less sparse, but not completely sparse, then the analysis remains correct, but the algorithms, as they now stand, produce results in which the relatively few like terms have not been collected.

We have said that the computing time will be analysed for completely sparse polynomials with integer coefficients; actually, it is only necessary that the coefficients be 'non-growing', to allow us to assume that the cost of coefficient arithmetic is constant. One way to guarantee this is to take the coefficients from a finite field. In this way, multiplication and addition costs do not grow with the number of digits in the result. Alternatively, one may suppose that all coefficient arithmetic is single-precision floating-point or single-precision integer arithmetic. To fix the model, we choose integer arithmetic.

The proposed cost model implies that the algorithm be analysed in terms of the number of coefficient multiplications required to compute the final result. This model will now be justified. The cost of an algorithm may be taken to be the number of elementary operations which occur during the computation. When the overhead is minimal, this reduces to the number of elementary arithmetic operations. The cost of multiplying two sparse polynomials is the cost of multiplying each term of one by each term of the other, and then collecting any like terms which may have been formed. Since the sparsity of the polynomials implies that the number of like terms is small, the cost of addition in actually collecting the like terms is negligible. This is for the general case. For the new algorithms presented in this thesis, wholly based on the principle of binomial expansion, no like terms are formed through the multiplication of sparse polynomials. Hence, there are no like terms to be collected.

To see this last point, consider computing  $f^n$  as

$$f_1^n + f_2^n + \sum_{r=1}^{n-1} \binom{n}{r} f_1^r f_2^{n-r},$$

where  $f_1$  and  $f_2$  are the two halves of  $f$ .

Imagine that the powers of both  $f_1$  and  $f_2$  have been computed. It can be shown that if any of the terms in the products of the summation combine, then  $f$  is not completely sparse to power  $n$ , contradicting the hypothesis. This will be proven below. In all of the algorithms presented in this

thesis, the fundamental arithmetic work consists, either of multiplying a binomial coefficient by a subpolynomial, or of multiplying a subpolynomial by another subpolynomial and then not collecting like terms, which do not exist in the completely sparse case. The total computation, therefore, consists of some number of monomial multiplications (coefficient multiplication plus exponent set addition) and some number of binomial coefficient multiplications (coefficient multiplication without exponent set addition.) The sum of these two numbers is the number of coefficient multiplications required by the algorithm, and also the number we use to measure the cost of the algorithm.

There is empirical evidence, as well as theoretical arguments, in favour of the cost model adopted in this thesis. Richard Fateman [5] has made theoretical and experimental analyses of algorithms for multiplication and powering of dense symbolic polynomials. These algorithms involved multiplications, divisions, additions, subtractions, and exponentiations. Comparing results of timings with actual counts of the above operations showed that actual computation times were closely proportional to the total of these counts and reasonably proportional to just the multiplications/divisions. In the binomial-expansion algorithms developed in this thesis, the only elementary operations are (1) product of monomial times monomial, and (2) product of binomial coefficient times monomial. For each of the algorithms presented, an exact count is given of the number of such products which occur.

This thesis is intended as a contribution to theoretical computational complexity. It may not be inappropriate, however, to stress again the practical concerns which underlie much work in symbolic algebraic manipulation. Symbol manipulation systems, such as the MACSYMA system at Project MAC or the Altran symbolic manipulator at Bell Telephone Laboratories, were developed as practical, cost-effective systems for actual computation. The designers of such systems were interested in the analysis of algorithms because they wanted sufficient understanding of the relative merits of competing algorithms to make concrete, practical decisions, viz., which algorithms to implement as system programs most suitable to the actual computations to be undertaken by their system. In this context, therefore, the design and analysis of algorithms has roots in practical problems existing, in some sense, outside of computer science.

Several algorithms for computing integer powers of sparse polynomials on sequential machines have been given recently by Fateman [4], and represent the state-of-the-art prior to the current work. The principal aims of this thesis are severalfold: (1) to exhibit superior algorithms for computation on sequential machines, (2) to provide insight, based on analysis and leading to intuition, as to why certain algorithms are better than others, (3) to make an exhaustive complexity analysis, both with respect to time and with respect to space, (4) to devise special-purpose computer architectures for parallel computation, and (5) to modify and



adapt the sequential algorithms to obtain parallel algorithms especially suited to the special-purpose computer architectures. The best parallel algorithms are then compared with the best sequential algorithms, to obtain the speed-up ratio, or the ratio of computation times on sequential and parallel machines.

Each of the algorithms presented in this thesis, whether for computation on a sequential machine or for computation on a parallel machine (two examples are provided in this thesis), takes a binomial-expansion approach to the symbolic computation of integer powers of sparse polynomials, as this approach is judged optimum. An exhaustive analysis of the whole family of binomial-expansion algorithms is undertaken. In this problem area, it is difficult, if not impossible, to prove that any one algorithm is optimal, even given a complete specification of the computational model, the cost model, and the general approach to be taken. However, in all of the binomial-expansion algorithms that we have been able to imagine, there is one, or perhaps two, which are vastly superior to all the rest. It is unlikely that the leading terms of their cost functions can be bettered. In the parallel case, we have two parallel algorithms, to be run on two special-purpose parallel architectures, both of whose speed-up ratios approach the theoretical upper limit in an asymptotic sense. In an even stronger asymptotic sense, the complexity of the sequential algorithms approaches the theoretical lower limit.

The family of sequential binomial-expansion algorithms will be explored systematically. According to Dijkstra [3]:

'A program should be conceived and understood as a member of a family ...'

Compare Parnas [13]:

'The aim of the new design methods is to allow the decisions which can be shared by a whole family, to be made before those decisions, which differentiate family members.'

Trees are convenient representations of program family structures. The root is the problem to be solved, and the terminal nodes are the fully-specified algorithms for doing so. The branches which descend from a node are the alternative design decisions which may be taken at that point. Such trees allow great insight into the relative costs of algorithms. In the parallel case, parallel binomial-expansion algorithms are developed for multiprocessor and associative processor architectures. The parallel algorithms and architectures are compared with each other, and with their sequential counterparts. In reality, we favour the associative processor architecture, even though its *prima facie* speed-up ratio is less. It was not possible to implement the parallel algorithms because the envisaged parallel machines have not been built yet.

In what follows, after collecting certain analytical results, there will be (1) a statement of the problem before the current work was undertaken, i.e., a survey of previous

results, (2) an exploration of the family of sequential binomial expansion algorithms, i.e., a systematic enumeration of design alternatives, (3) descriptions of the sequential algorithms, (4) analysis of how to minimize time in the sequential case, (5) analysis of how to minimize space in the sequential case, (6) discussion of the criteria by which the success and efficiency of parallel algorithms and computer systems are judged, (7) descriptions of special-purpose multiprocessor and associative-processor architectures, (8) descriptions of the parallel algorithms, (9) analysis of the speed-up ratio and space complexity of the parallel algorithms, and finally, (10) some comments about actual implementations of the most successful sequential binomial-expansion algorithm. In all that follows, a balance will be sought between analytic detail and informative general statements which provide a broad perspective.

A number of the new results appearing in this thesis have already been published in [2]. In large measure, the first five chapters of the thesis are an elaboration of sections from the earlier paper. In particular, most of the new results in Chapters IV and V appear already in [2]. The exceptions are the closed form for  $E(t, n)$ , and the entire discussion of BINF, unknown at the time the earlier paper was submitted. The new results in Chapters VI, VIII and IX have not yet been submitted for publication, and appear here for the first time.

## CHAPTER II

### NOTATION AND ELEMENTARY RESULTS

## CHAPTER II.

## NOTATION AND ELEMENTARY RESULTS

In this section we will state a number of elementary mathematical results, theorems, notations, and closed form expressions which will be of considerable use in subsequent analysis. The following standard theorems are quite useful.

$$\text{Theorem 2.1} \quad \sum_{i=0}^n \binom{r+i}{r} = \binom{r+n+1}{r+1} = \binom{r+n+1}{n}$$

$$\text{Theorem 2.2} \quad \sum_{i=0}^n \binom{r+i}{r} \binom{r+n-i}{r} = \binom{2r+n+1}{n}$$

**Theorem 2.3** Let  $f$  be a  $t$ -term polynomial which is completely sparse to power  $n$ . If  $\text{size}(f)$  gives the number of non-zero terms in  $f$ , then

$$\text{size}(f^n) = \binom{t+n-1}{n}, \quad t \geq 2$$

The proof is given, e.g., in Fateman [4].

$$\text{Theorem 2.4} \quad \sum_{i=1}^{n-1} \binom{r+i}{r} \binom{r+n-i-1}{r-1} = \binom{2r+n}{2r} - \binom{r+n-1}{r} - \binom{r+n}{r}$$

The proof is not difficult.

**Theorem 2.5** The multiplication of a polynomial  $f$  by a polynomial  $g$  can be done with a cost of  $\text{size}(f) \cdot \text{size}(g)$ .

Proof:

If  $f$  and  $g$  are two sparse polynomials, then the cost of obtaining their product is the cost of multiplying each term of one by each term of the other, and then collecting the like terms in the product so formed. (The assumption here, for completely sparse polynomials, is that straightforward classical multiplication is difficult to improve upon.) Inasmuch as the sparsity of polynomials implies that the number of like terms is small, the cost of addition in actually collecting the like terms is negligible. The cost, therefore, may be taken as  $\text{size}(f) \cdot \text{size}(g)$ , if the cost of the merge sort is ignored. More rigorously, if it is known a priori that there are no like terms, then the cost will be  $\text{size}(f) \cdot \text{size}(g)$ .

Consider the fact that

$$\sum_{i=1}^{n-1} \binom{t/2+i-1}{t/2-1} \binom{t/2+n-i-1}{t/2-1} = \binom{t+n-1}{n} - 2 \binom{t/2+n-1}{n}$$

This implies, in

$$f^n = (f_1 + f_2)^n = f_1^n + f_2^n + \sum_{i=1}^{n-1} \binom{n}{i} f_1^i f_2^{n-i},$$

where  $\text{size}(f_1) = \text{size}(f_2) = t/2$ , that no terms in any product may combine, because, if they did,  $f^n$  would no longer have the proper size for a completely sparse polynomial.

Let us review the notation for Stirling numbers of the first kind. Our reference here is Knuth [12].

TABLE 2.1 VALUES OF A FEW STIRLING NUMBERS OF THE FIRST KIND

	$\begin{bmatrix} n \\ 1 \end{bmatrix}$	$\begin{bmatrix} n \\ 2 \end{bmatrix}$	$\begin{bmatrix} n \\ 3 \end{bmatrix}$	$\begin{bmatrix} n \\ 4 \end{bmatrix}$	$\begin{bmatrix} n \\ 5 \end{bmatrix}$
$n = 1$	1	0	0	0	0
$n = 2$	1	1	0	0	0
$n = 3$	2	3	1	0	0
$n = 4$	6	11	6	1	0
$n = 5$	24	50	35	10	1

The following are important results concerning these numbers.

$$\begin{bmatrix} n \\ 1 \end{bmatrix} = (n-1)! \quad \begin{bmatrix} n \\ n \end{bmatrix} = 1 \quad \begin{bmatrix} n \\ n-1 \end{bmatrix} = \binom{n}{2} = \frac{1}{2} n(n-1)$$

$$\begin{bmatrix} n \\ m \end{bmatrix} = (n-1) \begin{bmatrix} n-1 \\ m \end{bmatrix} + \begin{bmatrix} n-1 \\ m-1 \end{bmatrix}, \quad n > 0$$

E.g.,

$$s(s+1)(s+2) = 2s + 3s^2 + s^3$$

$$= \sum_{j=1}^3 \begin{bmatrix} 3 \\ j \end{bmatrix} s^j$$

In general,

$$s(s+1)(s+2) \dots (s+p) = \sum_{j=1}^{p+1} \begin{bmatrix} p+1 \\ j \end{bmatrix} s^j \quad (2.6)$$

Let us now state and prove some preliminary closed forms of useful expressions which arise later in the analysis of these algorithms.

The function  $\text{size}(f)$  has already been introduced. Let us change the notation slightly so that  $\text{size}(s, n)$  is  $\text{size}(f^n)$  when  $\text{size}(f) = s$ . As usual, sparsity is assumed. We have

$$\text{size}(s, n) = \binom{s+n-1}{n} = \frac{1}{n!} \sum_{j=1}^n \begin{bmatrix} n \\ j \end{bmatrix} s^j \quad (2.7)$$

Proof:

$$\begin{aligned} \binom{s+n-1}{n} &= \frac{(s+n-1)!}{n!(s-1)!} \\ &= \frac{1}{n!} (s+n-1) \dots (s) \\ &= \frac{1}{n!} \sum_{j=1}^n \begin{bmatrix} n \\ j \end{bmatrix} s^j \quad \text{by (2.6)} \end{aligned}$$

We need a notation for the size of a collection or group of powers. Let  $\text{group}(s, n)$  be

$$\sum_{j=1}^n \text{size}(s, j)$$

i.e., the sum of sizes of all powers from 1 to  $n$  of a polynomial whose size is  $s$ . We have



$$\text{group } (s, n) = \binom{s+n}{n} - 1 = \frac{1}{n!} \sum_{j=1}^n \binom{n+1}{j+1} s^j \quad (2.8)$$

Proof:

$$\begin{aligned} \sum_{j=1}^n \binom{s+j-1}{j} &= \binom{s+n}{n} - 1 \\ &= \frac{(s+n)!}{n! s!} - 1 \\ &= \frac{1}{n!} \cdot \frac{1}{s} \cdot (s+n) \dots (s) - 1 \\ &= \frac{1}{n!} \cdot \frac{1}{s} \cdot \sum_{j=1}^{n+1} \binom{n+1}{j} s^j - 1 \quad \text{by (2.6)} \\ &= \frac{1}{n!} \sum_{j=1}^n \binom{n+1}{j+1} s^j, \end{aligned}$$

since

$$\binom{n+1}{1} = n!,$$

where  $j$  has been replaced by  $j+1$ . The coefficient of the leading term, viz.,  $\binom{n+1}{n+1}$ , is 1.

Another useful closed-form expression is the following.

$$\binom{s+n}{n-1} - n = \frac{1}{(n-1)!} \sum_{j=1}^n \sum_{k=0}^{n-j-1} \binom{n-1}{k+j} \binom{k+j}{k} 2^k s^j \quad (2.9)$$

where, as a convention,  $\sum_{k=0}^{-1} \dots$  is zero.

Proof:

$$\begin{aligned}
 \binom{s+n}{n-1} - n &= \frac{(s+n)!}{(n-1)!(s+1)!} - n \\
 &= \frac{1}{(n-1)!} (s+n) \dots (s+2) - n \\
 &= \frac{1}{(n-1)!} \sum_{j=1}^{n-1} \begin{bmatrix} n-1 \\ j \end{bmatrix} (s+2)^j - n \quad \text{by (2.6)} \\
 &= \frac{1}{(n-1)!} \sum_{j=1}^{n-1} \begin{bmatrix} n-1 \\ j \end{bmatrix} \sum_{k=0}^j \binom{j}{k} s^k 2^{j-k} - n \\
 &= \frac{1}{(n-1)!} \sum_{j=1}^{n-1} \begin{bmatrix} n-1 \\ j \end{bmatrix} \sum_{k=1}^j \binom{j}{k} s^k 2^{j-k},
 \end{aligned}$$

since the constant term in  $(s+n) \dots (s+2)$  is  $n!$ .

Expansion and collection of like terms gives finally

$$\binom{s+n}{n-1} - n = \frac{1}{(n-1)!} \sum_{j=1}^n \sum_{k=0}^{n-j-1} \begin{bmatrix} n-1 \\ k+j \end{bmatrix} \binom{k+j}{k} 2^k s^j,$$

where the term for  $j=n$  contributes nothing, as explained above.

The closed-form expressions which have just been introduced are instrumental in obtaining a closed-form expression for the time complexity of the best sequential binomial-expansion algorithm. The function  $\text{size}(s, n)$  is used continually in sparse polynomial time complexity analysis; together with the function  $\text{group}(s, n)$  it also gives an important tool for sparse polynomial space complexity analysis.

Finally, there is a theorem, suggested by Michael Fredman [4], which gives a lower bound on the number of multiplications required to compute a power of a polynomial. That number is at least equal to  $\text{size}(f^n) - \text{size}(f)$ .

**Theorem 2.10** No algorithm can compute  $f^n$ , the  $n^{\text{th}}$  power of an arbitrary polynomial, in fewer than  $\text{size}(f^n) - \text{size}(f)$  multiplications.

The proof may be found in Fateman [4]. Suppose now that  $f$  is completely sparse, and that  $\text{size}(f) = t$ . According to Theorem 2.10, the lower limit on the number of multiplications required by any algorithm to compute  $f^n$ , which will be denoted by  $L(t, n)$ , is given by

$$\begin{aligned} L(t, n) &= \text{size}(t, n) - t = \binom{t+n-1}{n} - t \\ &= \frac{1}{n!} \sum_{j=1}^n \binom{n}{j} t^j - t \quad \text{by (2.7)} \end{aligned}$$

This formula may also be written as

$$L(t, n) = \frac{t^n}{n!} + \frac{t^{n-1}}{2(n-2)!} + O(t^{n-2})$$

This is the least number of multiplications possible. When one has a good algorithm, one compares the number of multiplications it uses with the theoretical lower limit to see how far off one is. The cost functions for BINE and BINF,

these latter being the two best sequential binomial-expansion algorithms presented in this thesis, express the number of multiplications used by these algorithms. The leading terms of either cost function are

$$\frac{t^n}{n!} + t^{n-1} \left[ \frac{1}{2(n-2)!} + \frac{1}{2^{n-2}(n-1)!} \right] + o(t^{n-2}),$$

for  $n > 2$ . We will show by comparison with the cost functions of other algorithms just how good this is.

### CHAPTER III

#### STATE OF THE PROBLEM PRIOR TO THE PRESENT WORK

## CHAPTER III

STATE OF THE PROBLEM PRIOR TO THE  
PRESENT WORK

The best previously-existing sequential algorithm for computing powers of sparse polynomials is due to Fateman [4]. It is one of several algorithms for this problem he analysed to obtain insight into their relative merits. We shall reproduce his descriptions of these algorithms and adopt his analyses, except in the case of his best algorithm, which will be analysed anew. Another computationally much less interesting, i.e., much more expensive, algorithm for computing powers of symbolic polynomials, due to Horowitz and Sahni, will also be mentioned. Fateman has considered repeated multiplication (RMUL), repeated squaring (RSQ), two specific approaches based on multinomial expansion (NOMA and NOMB), and two specific approaches based on binomial expansion (BINA and BINB), all as methods for computing integer powers of sparse polynomials. The main conclusion is, that of these six algorithms, algorithm BINB is computationally most efficient. In general, the analysis suggests that binomial expansion is the most promising general approach. This analysis led to a research programme to develop those superior binomial-expansion algorithms for serial and parallel computation which are presented in this thesis.

### 3.1 ALGORITHM RMUL (REPEATED MULTIPLICATION)

#### Description:

RMUL successively computes  $f^2 = f.f$ ,  $f^3 = f.f^2, \dots$ ,  $f^n = f.f^{n-1}$ . Cf. Gentleman [7].

#### Analysis:

By Theorem 2.5, the cost for the polynomial multiplications is

$$\begin{aligned} \text{size}(f) \cdot \sum_{i=1}^{n-1} \text{size}(f^i) &= t \cdot \sum_{i=1}^{n-1} \binom{t+i-1}{t-1} \\ &= t \cdot \binom{t+n-1}{t} - t \quad (3.1.1) \end{aligned}$$

by Theorems 2.3 and 2.1.

### 3.2 ALGORITHM RSQ (REPEATED SQUARING)

#### Description:

When  $n$  is a power of 2, RSQ computes  $f^n$  as  $f^2 = f.f$ ,  $f^4 = f^2.f^2, \dots$ ,  $f^n = f^{n/2}.f^{n/2}$ . When  $n$  is not a power of 2, it computes a sequence of powers of  $f$  based on the expansion of  $n$  as a binary number. More formally

- (1)  $q \leftarrow 1$ .  $z \leftarrow f$ .
  - (2)  $i \leftarrow$  rightmost bit of  $n$ . Shift  $n$  right one bit.
- If  $i = 0$ , go to (4).

- (3) If  $q = 1$ ,  $q \leftarrow z$  else  $q \leftarrow q \cdot z$ .
- (4) If  $n = 0$ , return  $q$  else  $z \leftarrow z \cdot z$  and go to (2).

That this algorithm correctly computes  $f^n$  by means of binary expansion of the power  $n$  may be seen by considering  $n$  as a binary number and observing the relationship between multiplication of powers and addition of exponents.

#### Analysis:

The analysis is done for  $n$  a power of 2. Even in this case, RMUL is superior to RSQ. When  $n$  is not a power of 2, this superiority (of RMUL) merely increases. When  $n = 2^j$ , the cost for RSQ is

$$\sum_{i=0}^{j-1} \text{size}(f^{2^i})^2 = \sum_{i=0}^{j-1} (t+2^i-1)^2 \quad (3.2.1)$$

For  $n = 4$ , RSQ costs  $\frac{1}{4}(t^4 + 2t^3 + 5t^2)$  whereas RMUL costs  $\frac{1}{6}(t^4 + 6t^3 + 11t^2)$ . For  $t > 7$ , RMUL is cheaper. For  $n = 8$ , RMUL is better for  $t > 3$ . Both Fateman [4] and Gentleman [7] discuss the nature of the superiority of RMUL to RSQ. Fateman remarks that it is less costly to multiply a large polynomial by a small polynomial, than to multiply two polynomials of intermediate size.



### 3.3 ALGORITHM NOMA (FULL MULTINOMIAL EXPANSION - A)

#### Description:

Let  $f = a_1 + \dots + a_t$  where the  $a_i$ ,  $1 \leq i \leq t$ , are monomials. Assume that we have precomputed the  $n^{\text{th}}$  power of  $g = \alpha_1 + \dots + \alpha_t$  where the  $\alpha_i$  are new symbols (not in  $f$ ). Then all we need to do is to substitute  $a_i$  for  $\alpha_i$  in  $g^n$ . We can compute the substitutions using Horner's rule. For the detailed algorithm description, refer to Fateman's paper [4].

#### Analysis:

The cost which is obtained is

$$\sum_{j=0}^n t \binom{t+j-2}{t-1} = t \binom{t+n-1}{t} \quad (3.3.1)$$

### 3.4 ALGORITHM NOMB (FULL MULTINOMIAL EXPANSION - B)

#### Description:

For each of the  $t$  monomials in  $f$ ,  $a_1, \dots, a_t$ , compute a list  $\ell_i = \langle a_i^0, a_i^1, \dots, a_i^n \rangle$ . Next, compute all products consisting of combinations of no more than one element  $a_i^{n_i}$  from each list such that  $n_1 + \dots + n_t = n$ . This product is then multiplied by the multinomial coefficient

$$\binom{n}{n_1, \dots, n_t}$$

For more details, see Fateman [4].

Analysis:

The cost which is given is

$$t \binom{t+n-2}{t-1} + tn - 2t \quad (3.4.1)$$

3.5 ALGORITHM BINA (BINOMIAL EXPANSION WITH MONOMIAL SPLITTING)

Description:

The polynomial  $f$  is split into two parts  $f_1 + f_2$ . The size of  $f_1$  is 1, and the size of  $f_2$  is  $t-1$ . To compute  $f^n$ , we compute  $f_1^2, \dots, f_1^n$  and  $f_2^2, \dots, f_2^n$ , and use the binomial theorem

$$f^n = \sum_{i=0}^n \binom{n}{i} f_1^i f_2^{n-i}$$

Note that  $f_2^2$  can be calculated by BINA, and other powers are simply computed by  $f_2^i = f_2 \cdot f_2^{i-1}$ . See Fateman [4].

Analysis:

Itemizing the steps and their costs, and adding gives finally the cost for BINA,  $n \geq 2$ . It is

$$t \binom{t+n-2}{t-1} - \frac{1}{2}(t^2 - 3t + 10) + 2n \quad (3.5.1)$$

### 3.6 ALGORITHM BINB (BINOMIAL EXPANSION WITH HALF SPLITTING)

#### Description:

BINB is identical in concept to BINA, but  $f$  is split as evenly as possible into  $f_1 + f_2$ . See Fateman [4].

#### Analysis:

When Fateman itemizes the steps and their costs, and adds, he obtains as the cost for BINB

$$\begin{aligned} & \binom{t+n-1}{n} + t \binom{t/2+n-1}{n-1} - 2 \binom{t/2+n-1}{n} - t/2(t/4-n- \\ & - \log_2(t-1)+4) \end{aligned} \quad (3.6.2)$$

We repeat the entire analysis below and obtain a slightly higher cost function for BINB.

RMUL is clearly cheaper than RSQ. Comparing (3.1.1) with (3.3.1) we see that, surprisingly, NOMB is no better than RMUL. Comparing (3.4.1) with (3.1.1), we see that NOMB/RMUL is approximately  $t/(t+n-1)$ . That is, for a given  $t$ , NOMB gets arbitrarily better as  $n$  increases. Yet NOMB is difficult to program. Comparing (3.5.1) with (3.1.1) and (3.4.1) gives: BINA/NOMB is approximately 1, and BINA/RMUL is approximately  $t/(t+n-1)$ . Again, for fixed  $t$  and increasing  $n$ , BINA becomes arbitrarily better than RMUL. Finally,

comparing (3.6.2) with (3.5.2), and calculating the leading terms in both cases, we see that BINB is consistently better than BINA. The leading term of the latter is  $t^n/(n-1)!$ ; the leading term of the former, according to Fateman [4], is

$$t^n \left[ \frac{1-2^{1-n}}{n!} + \frac{2^{1-n}}{(n-1)!} \right].$$

These cost functions demonstrate clearly, that among these six algorithms, algorithm BINB is the most computationally efficient. Questions of efficiency and ease of programming make the binomial-expansion algorithms appear as good choices for computing powers of sparse polynomials. Fateman conjectures that an algorithm superior to BINB may be hard to find. The conjecture is, of course, false. Yet BINB is not, by any means, a bad algorithm insofar as it does approach the theoretical lower limit for coefficient multiplications when  $t$  and  $n$  become large. In this context, the multinomial-expansion algorithm for powering sparse polynomials proposed by Horowitz and Sahni[10] is considerably less interesting. Their cost function is only of the order of the theoretical lower limit, in one case being roughly equal to four times that limit. It is possible to interpret this result as even further evidence for the superiority of the binomial-expansion approach.

CHAPTER IV  
THE ALGORITHM FAMILY

## CHAPTER IV

### THE ALGORITHM FAMILY

Professor Dijkstra has been quoted above to the effect that:

"A program should be conceived and understood as a member of a family ..."

The reasoning here, relevant both to algorithm design and to algorithm analysis, is that choices, the design decisions taken in any algorithm, can be evaluated only if one sees them against the background of a range of possible alternatives. One evaluates a specific design decision which is taken in comparison with other design decisions which could have been taken. Ultimately, one evaluates a specific algorithm which is proposed in comparison with other algorithms which could have been proposed. In this way, one comes to understand the structure of a family of algorithms which solve the same problem. A tree structure, branching downwards from the root, is an extremely convenient graphical representation of the algorithm family structure, of its decision points and possible decisions. The root of the tree represents the problem to be solved. The terminal nodes represent fully-specified algorithms, some better, some worse, which solve the problem. In general, each non-terminal node of the tree represents a point at which a decision must be taken, a choice situation, while the branches which descend downwards from that node represent the possible decisions at that point, in that situation.

At the root of the program family tree may be placed a description or specification of the problem to be solved. In our case it is, construct an algorithm which accepts an input power  $n$ ,  $n$  a positive integer, and an input polynomial  $f$ ,  $f$  completely or almost completely sparse to power  $n$ , and produces as its output the resultant polynomial  $f^n$ . As mentioned previously, only a sub-family, namely the family of sequential binomial-expansion algorithms, will be systematically explored using the program family tree. Binomial-expansion algorithms are considered the most promising; parallel algorithms will be considered later in the thesis. The chief analytic aim associated with the program family tree is to make statements about the relative costs of the various design decisions associated with different branching points over and above the analytically-obtained cost functions which allow us to choose among the fully-specified algorithms on the basis of their time complexity. A knowledge of these relative costs allows us to explain the superiority (least time-cost) of one particular algorithm as having resulted from consistently lowest-cost decisions. It also adds weight to the conjecture that this particular algorithm, the best so far, is optimal. This optimality is guaranteed if no superior design decision exists at any branching point; it will be destroyed if and when someone succeeds in imagining a design decision superior to those considered here.

For a while, attention will be restricted to, and arguments concern, the sequential case. When attention is redirected to the parallel case, the parallel designer must be prepared to revise any decision which, although clearly correct in the sequential case, is likely to hold back parallelism in the parallel case. There is a body of analysis, confirmed by results obtained here, which suggests that binomial-expansion is superior to other general approaches used in symbolic powering of sparse polynomials. This analysis has been done for the sequential case; no one has analytic results which suggest that binomial expansion is also the most promising parallel approach. The parallel algorithms developed in this thesis are binomial-expansion algorithms, the chief reason for this being primarily the extreme attractiveness of the sequential binomial-expansion algorithms. Although the speed-up ratios of the parallel algorithms both approach the theoretical upper limit, this is no guarantee of optimality of approach in the parallel case. With respect to the design decisions within the binomial-expansion sub-family, it has generally been found that those decisions which minimize computing time in the sequential case do not interfere with the possibilities of exploiting parallelism in the parallel case. The very good parallel algorithms, then, make use of the ideas which have been developed for the best sequential algorithms.



## 4.1 DESIGN DECISIONS

In this section, we explain the four major design decision areas which arise when one considers algorithms for computing integer powers of sparse polynomials. The program family tree is a graphical accompaniment which records the design decisions which differentiate algorithms of the family, and so serves to relate these algorithms to each other.

### 4.1.1 General Approach

Every algorithm proposed in this thesis takes a binomial expansion approach to computing integer powers of sparse polynomials. This means that if one has an input polynomial  $f$ , which is to be raised to some power  $n$ , one splits  $f$  into two subpolynomials  $f_1$  and  $f_2$  (which together, contain all the monomials of the original polynomial), obtains the appropriate powers of the subpolynomials, and then combines the subpolynomial powers together with binomial coefficients according to the theorem of binomial expansion to obtain the final answer,  $f^n$ . That is,  $f^n$  is computed as

$$f^n = (f_1 + f_2)^n = \sum_{r=0}^n \binom{n}{r} f_1^r f_2^{n-r}$$

Binomial expansion has been chosen in preference to some other general approach, such as repeated multiplication, repeated squaring, or some form of multinomial expansion, not because every binomial-expansion algorithm is superior to every non-

binomial-expansion algorithm, but rather because the best binomial-expansion algorithms are superior to the best algorithms in each of the other general approaches. The position here is that binomial expansion allows a lower cost refinement in comparison with the refinements available in other approaches. As a graphical convention in connection with the program family tree, we agree to draw the branch which represents the least cost, or possibility of least cost, on the left. The first two branches of the tree, then, which descend from the root, are 'binomial expansion' on the left, and 'some other approach' on the right. The binomial-expansion family is extremely rich in algorithms. BINA and BINB, the two previously-published binomial-expansion algorithms, are only two of many possible refinements. These refinements differ considerably in cost. It is the position here that BINB, the best previously-published binomial-expansion algorithm, does not take optimal design decisions by a wide margin. The obvious strategy to improve on BINB, indeed, to look for the optimal algorithm, is to consider a broad range of possible refinements of this most attractive general approach.

#### 4.1.2 Splitting

In binomial expansion the original polynomial  $f$  is split into two parts,  $f_1$  and  $f_2$ . The relative sizes of  $f_1$  and  $f_2$  affect the time complexity of a binomial-expansion algorithm in two distinct and unrelated ways: on the one hand

they affect the time complexity of generating powers of subpolynomials, and on the other hand they affect the time complexity of combining appropriate generated powers, suitably multiplied by binomial coefficients, in the binomial expansion. These effects are so large that an algorithm with optimal splitting and a poor technique for generating powers may outperform another algorithm with suboptimal splitting and a good technique for generating powers. Some of the ways of splitting polynomials into two parts are: (a) one term and the rest, (b) as evenly as possible, (c) as unevenly as possible, a power of two and the rest, when the number of terms to be split is not a power of two, else evenly, and finally, (d) as evenly as possible, a power of two and the rest. Considering many ways of splitting is consistent with the idea of investigating a broad range of possible refinements. We found that degree of evenness of splitting is the only relevant splitting parameter in choosing from among the four splittings mentioned.

The need to choose the relative sizes of the subpolynomials  $f_1$  and  $f_2$ , where  $f = f_1 + f_2$ , follows from the very idea of binomial expansion. There is another splitting decision, however, which arises only if one considers alternative ways of generating powers of subpolynomials. The two binomial-expansion algorithms BINA and BINB adopt repeated multiplication as their principal strategy for computing powers of subpolynomials. It is characteristic of this approach that the original polynomial  $f$  is split only once; we call this

characteristic: 'single-level splitting'. Single-level splitting is a consequence of a 'non-recursive' approach to the problem, in which binomial expansion is used to compute the power of the whole polynomial, and something entirely different is used to compute the powers of subpolynomials. If, on the other hand, one carries through and uses binomial expansion to compute the powers of subpolynomials, one needs to split the original polynomial over and over again, until finally the monomial level is reached. This is called: 'multilevel splitting'. All the algorithms presented in this thesis use variants of binomial expansion for generating powers of subpolynomials, and hence are committed to multilevel splitting, in contrast to previously-published binomial-expansion algorithms, which all adopt the single-level splitting approach.

Analysis of the fully-specified algorithms allows one to make two assertions about the preferred design decisions relating to splitting. It is less costly to multiply a large polynomial by a small polynomial, than to multiply two polynomials of intermediate size. It follows that even splitting increases the cost of combining computed powers of subpolynomials. All the techniques for computing subpolynomial powers, whether variants of binomial expansion or simply repeated multiplication, are such that even splitting reduces the cost of computing these powers. In all instances, the latter effect predominates. Hence, in either a single-level or a multilevel context, polynomial splitting which is as even as

possible is always to be preferred. This is the first assertion. From the standpoint of time complexity, repeated multiplication is a poor way to compute powers of subpolynomials. Both the variants of binomial expansion, to be discussed shortly, easily outperform repeated multiplication for this task. Hence, in comparison with single-level splitting, multilevel splitting is to be preferred. And this is the second assertion.

Of two sequential algorithms which are equally parallelizable, that is, possess identical speed-up ratios, the least-cost sequential algorithm will give rise to the least-cost parallel algorithm. There is no evidence which suggests that even splitting reduces the possibilities for parallelism. On the contrary, even splitting breaks problems into subproblems of equal sizes, which can then be processed simultaneously. One may also say that even splitting is fully consistent with trying to maximize the number of independent subtasks, an important step in extracting parallelism from a sequential algorithm. Repeated multiplication, and hence single-level splitting, has the following disadvantage in the parallel case. Because, even for completely sparse polynomials, there is no theorem which asserts that no like terms are formed when generating subpolynomial powers through repeated multiplication, this latter technique is less easily parallelizable (because of the need to collect like terms) than the multilevel binomial-expansion techniques, for which it can be shown, as discussed previously, that no like terms are formed in the

cross products of the binomial expansion. Thus far at least, the same decisions seem reasonable in both the sequential and parallel cases; no further claim is made.

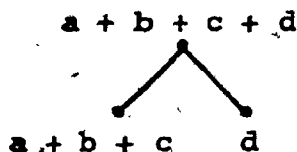
### 1.3 Powers of Subpolynomials

Powers of subpolynomials must first be computed before they can be combined in the binomial expansion together with binomial coefficients. What strategy best computes these subpolynomial powers is one of the least immediately apparent issues in the whole binomial-expansion family, and yet one central to the time complexity. Previous binomial-expansion algorithms have essentially used repeated multiplication for this task; the new algorithms presented and analysed in this thesis use the techniques of recursion and binary merge. These techniques will now be explained. As soon as the decisions concerning depth (level) and evenness of splitting have been taken, the splitting of the original polynomial is fully specified, and may be displayed as a binary tree. A 'term group' is one or more terms from the original polynomial; the tree is called the 'term group tree'. The whole polynomial is placed at the root. New nodes (subnodes) are formed by taking the term group at a node, and splitting it into halves of appropriate sizes (determined by the evenness decision) to form the left and right subnodes. Convention: let the size of the term group in the left subnode never be smaller than that in the right subnode. When multilevel splitting has been adopted, this process is repeated until one reaches the individual terms,

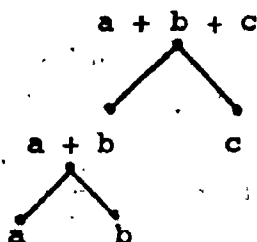
which are the terminal nodes of the term group tree.

Examples:

Single-level, one-and-the-rest splitting



Multilevel, as-even-as-possible splitting



Multilevel, even splitting with  $t = 2^k$

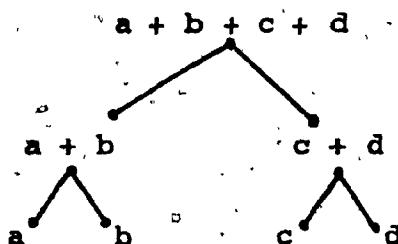


FIG. 4.1 THREE TERM-GROUP TREES

Recursion properly so-called and binary merge are the two 'recursive' approaches to the computation of powers of subpolynomials. They are consistent applications of binomial expansion to the computation of powers of polynomials at all stages, except, of course, at the monomial level itself. The idea of recursion is to obtain the powers of the subpolynomials by recursive application of the total algorithm to the subnodes. The weakness of this approach, still vastly superior to repeated multiplication, is that one never computes single powers of subpolynomials, but rather groups of all powers from two to  $n$ . Separate recursive application for distinct powers of a subpolynomial leads to recomputation: distinct powers of a binomial have computable factors in common. Binary merge is a form of recursion in which there is no recomputation. One may compute the power of a node, or the group of powers (from two to  $n$ ) of a node, if one has the groups of powers (from two to  $n$ ) of both subnodes. Equivalently, once one has computed groups of powers of subnodes, one may go on to compute groups of powers (or single powers) of the father node. In binary merge we progress up the tree, starting from the terminal nodes, using the already computed powers of the subnodes (once they have been computed) to compute the groups of powers of the father nodes. Ultimately, a single power of the root node, which contains the original polynomial, is computed.



#### 4.1.4 Combining Powers

We assume that a table of binomial coefficients  $\binom{r}{s}$ ,  $0 \leq s \leq r \leq n$ , is available to the computation. References to this table are overhead. According to the cost model, each multiplication of a monomial by a binomial coefficient counts as one coefficient multiplication. If the powers of the two subpolynomials  $f_1$  and  $f_2$ , where  $f = f_1 + f_2$ , are available,

$$f^n = (f_1 + f_2)^n = \sum \binom{n}{r} f_1^r f_2^{n-r}$$

may be obtained by combining suitable powers with binomial coefficients in the binomial expansion. The product  $\binom{n}{r} f_1^r f_2^{n-r}$  may be obtained either (a) left to right, irrespective of the relative sizes of  $f_1^r$  and  $f_2^{n-r}$ , or (b) by first multiplying  $\binom{n}{r}$  by the smaller of the two polynomials, and then this result by the remaining polynomial. Previous algorithms have invariably used the first technique; our best algorithms use the second technique. It is immediate that the second technique reduces the number of coefficient multiplications. We call the first technique 'left-to-right', and the second technique 'smaller'. If the first technique is used, the product may also be written as  $f_2^{n-r} \cdot \binom{n}{r} f_1^r$ . That is, one always multiplies by the power of  $f_1$  first. In the second technique, one chooses the order of multiplication for each product.

If one were to list the preferred choices which have been made so far, one would have, first, the use of binomial expansion, second, splitting the polynomial as evenly as possible, third, adopting one of the 'recursive' approaches, which commits one to multilevel splitting, fourth, preferring binary merge to recursion properly so-called, and last, forming the cross products using the smaller technique. These choices specify an algorithm, which we have called BINE, about which it is conjectured that the leading terms of the time-complexity cost function cannot be bettered. There is, however, one final improvement which can be made. Suppose that  $a$  and  $b$  are the left and right subnodes of the root of the term group tree, and that  $b_1$  and  $b_2$  are the left and right subnodes of  $b$ . We will use the same names for a node, and for the term group associated with that node. Suppose that we are computing the fourth power of the root. In that case, we are interested in the product  $a^2 \cdot 6b^2$ . It is not necessary to have previously computed  $b^2$  as  $b_1^2 + b_2^2 + b_1 \cdot 2b_2$ . Rather, one may compute  $6b^2$  directly as  $6b_1^2 + 6b_2^2 + b_1 \cdot 12b_2$ . One saves as many coefficient multiplications as there are terms in the product  $b_1 \cdot b_2$  at the cost of multiplying 6 by 2. This technique is called: 'distribution'; it amounts to not having to compute some of the lower powers of the left and right subnodes of the root. Details of the technique will be given below in Section 5.6, which analyses the time savings due to distribution. Distribution is a mixed strategy which deviates from pure binomial expansion at the highest levels of the term group tree. If the

algorithm which differs from BINE only in that distribution is used, be called BINF, then BINF is the least-cost sequential binomial-expansion algorithm (for computing powers of sparse polynomials) known at the time of writing.

#### 4.1.5 Optimum Design Decisions

If we think of distribution as a modified form of merge, then the whole set of optimum design decisions can be diagrammed in the following way:

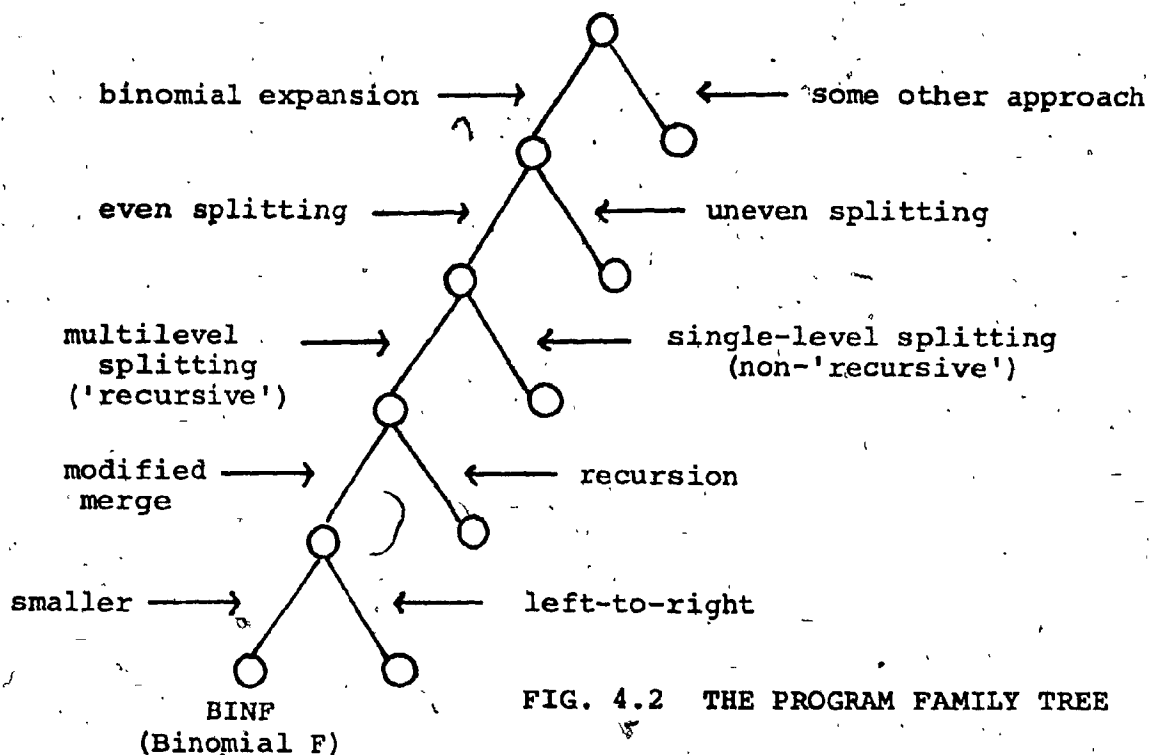


FIG. 4.2 THE PROGRAM FAMILY TREE

The diagram which has been given is a very partial and incomplete representation, even of the sub-family of binomial-expansion algorithms. The reader may imagine the combinatoric possibilities in permuting choices. Two algorithms will differ in cost if even one decision is not the same. In the

next section we present and analyse some representative algorithms, most of them new. The superiority of BINF will be demonstrated; if no leftmost branch above can be improved upon, then this algorithm is indeed optimal.

Before passing to the section for formal description and analysis of algorithms, we may use the program family tree for an initial comparison between previously existing binomial-expansion algorithms and those presented in this thesis. Obviously, all belong to the binomial-expansion subfamily. Fateman's BINA employs monomial splitting (the most uneven splitting), while his BINB splits as evenly as possible. Most of the new algorithms use even splitting; those that do not pay a time penalty. BINA and BINB are both non-recursive, single-level-splitting algorithms, as repeated multiplication generates the powers of subpolynomials. All the new algorithms are 'recursive', multilevel-splitting algorithms, using, variously, merge, modified merge, or recursion for the generation of subpolynomial powers. BINA and BINB use left-to-right. Many of the new algorithms use smaller; those that do not pay a time penalty. The program family tree is an important complement to the exact, analytically-obtained cost functions. It gives us a capsule view of the full range of differences between any two algorithms. It also provides a graphical representation of which decisions are to be preferred at each decision point. In this way, we come to understand why certain algorithms outperform others.

CHAPTER V

TIME COMPLEXITY OF ALGORITHMS  
(SEQUENTIAL OPTION)

## CHAPTER V

TIME COMPLEXITY OF ALGORITHMS  
(SEQUENTIAL OPTION)

We discuss below Fateman's BINB, which we analyse anew, and four algorithms which are all refinements of the binomial-expansion approach. Each of the four algorithms is superior to BINB. The algorithms will be described using the terminology developed in connection with the program family tree of Chapter IV. For each algorithm, the input is a power  $n$ , and a polynomial  $f$  completely sparse to power  $n$ , while the output is the resultant polynomial  $f^n$ . The aim of the time-complexity analysis is to give the cost in coefficient multiplications as a function of  $t$  and  $n$  for each algorithm.

5.1 ALGORITHM BINB (Binomial B)

This algorithm is specified by the design decisions: binomial expansion, even splitting, single-level splitting, insofar as recursion is used for the squares of subpolynomials and repeated multiplication for all other powers, and finally, left-to-right. The cost function proposed in [4] is

$$B(t, n) = \binom{t+n-1}{n} + t \binom{t/2+n-1}{n-1} - 2 \binom{t/2+n-1}{n} - t/2(t/4 - n - \log_2(t-1) + 4) \quad (5.1.1)$$

As stated, BINB uses a mixed strategy for computing powers of

subpolynomials: recursion for the squares, and repeated multiplication for higher powers. There may be no deep philosophy behind this; a remark by Fateman in another paper [5] shows he did not think it made much difference. After closely inspecting the steps in [4, p.152-153], we would itemize the total cost of this algorithm as follows:

Case 1:  $t = 2^k$   $f = f_1 + f_2$   $\text{size}(f_1) = \text{size}(f_2) = t/2$

TABLE 5.1 ITEMIZING THE COSTS IN BINB

Step	Compute	Cost
1	$f_1^2, f_2^2$	$t^2/4 + kt/2$
2	$f_1^3, \dots, f_1^n$ $f_2^3, \dots, f_2^n$	$2 \cdot t/2 \sum_{i=2}^{n-1} \binom{t/2+i-1}{t/2-1}$
3	$\binom{n}{1} f_2$	$t/2$
4	$\binom{n}{2} f_2^2, \dots, \binom{n}{n-1} f_2^{n-1}$	$\sum_{i=2}^{n-1} \binom{t/2+i-1}{t/2-1}$
5	$f_1^i \binom{n}{i} f_2^{n-i},$ $1 \leq i \leq n-1$	$\sum_{i=1}^{n-1} \binom{t/2+i-1}{t/2-1} \binom{t/2+n-i-1}{t/2-1}$

Total Cost

$$B_1(t, n) = \binom{t+n-1}{n} - 2 \binom{t/2+n-1}{n} + (t+1) \binom{t/2+n-1}{n-1}$$

$$- t/2(t/2 - \log_2 t + 1) - 1 \quad (5.1.2)$$

Case 2:  $t \neq 2^k$   $f = f_1 + f_2$   $\text{size}(f_1) = \lceil t/2 \rceil$   
 $\text{size}(f_2) = \lfloor t/2 \rfloor$

Itemizing the cost as before we get

$$B_2(t, n) = \binom{t+n-1}{n} - \binom{t_1+n-1}{n} - \binom{t_2+n-1}{n} + \\
+ t_1 \binom{t_1+n-1}{n-1} + (t_2+1) \binom{t_2+n-1}{n-1} + p(t) \quad (5.1.3)$$

where  $p(t)$  is a polynomial in  $t$  of degree 2.

Comparing (5.1.1) with (5.1.2) we note that, even for  $t = 2^k$ , the cost we calculate is greater than the reported cost. This difference will show up in the second leading term of the cost function  $B(t, n)$ .

## 5.2 ALGORITHM BINC (Binomial C)

This algorithm is specified by the design decisions: binomial expansion, even splitting, multilevel splitting, as recursion is used to generate all powers of all subpolynomials other than monomials, and finally, left-to-right. The analysis will be carried through for  $t = 2^k$ . The original polynomial  $f$  is split into two parts,  $f_1$  and  $f_2$ , where

$$\text{size}(f_1) = \text{size}(f_2) = t/2$$

We use the binomial theorem  $f^n = \sum_{r=0}^n \binom{n}{r} f_1^r f_2^{n-r}$  to compute  $f^n$ .



The powers of the subpolynomials (other than monomials) are obtained by recursive application of the algorithm just specified.

### 5.2.1 Analysis of BINC

Let  $C(t,n)$  be the cost in coefficient multiplications of computing  $f^n$  when  $\text{size}(f) = t$ . (The cost function is always the last letter of the algorithm name.) If we expand  $f^n$  in the form

$$f^n = f_1^n + f_2^n + \sum_{r=1}^{n-1} \binom{n}{r} f_1^r f_2^{n-r}$$

we have

$$C(t,n) = 2C(t/2,n) + \sum_{r=1}^{n-1} Q_r \quad (5.2.1)$$

where  $Q_r$  is the sum of costs itemized as follows:

TABLE 5.2 ITEMIZING THE COSTS IN BINC

Step	Compute	Cost
1	$f_1^r$	$C(t/2,r)$
2	$f_2^{n-r}$	$C(t/2,n-r)$
3	$\binom{n}{r} f_1^r$	$\binom{t/2+r-1}{t/2-1}$
4	$\binom{n}{r} f_1^r f_2^{n-r}$	$\binom{t/2+r-1}{t/2-1} \binom{t/2+n-r-1}{t/2-1}$

$$C(t,n) = 2C(t/2,n) + 2 \sum_{r=1}^{n-1} C(t/2,r) + \sum_{r=1}^{n-1} \binom{t/2+r-1}{t/2-1} + \sum_{r=1}^{n-1} \binom{t/2+r-1}{t/2-1} \binom{t/2+n-r-1}{t/2-1} \quad (5.2.2)$$

Using (2.1) and (2.2) we obtain

$$C(t,n) = \binom{t+n-1}{n} - 2 \binom{t/2+n-1}{n} + \binom{t/2+n-1}{n-1} - 1 + 2 \sum_{r=1}^n C(t/2,r) \quad (5.2.3)$$

We may bootstrap a closed form for  $C(t,n)$  into one for  $C(t,n+1)$  by using the formula:

$$C(t,n+1) = nt + \frac{t-1}{n+1} \binom{t+n-1}{n} + \sum_{r=1}^k 2^{r-1} \binom{t/2^r+n-1}{n} + \sum_{r=1}^k 2^{r-1} C(t/2^{r-1},n) \quad (5.2.4)$$

obtained from (5.2.3) by changing  $n$  to  $n+1$  and using recurrence on  $t$ . Using this formula and induction on  $n$ , we obtain the general form of  $C(t,n)$ .

$$C(t,n) = \sum_{i=1}^n a_i^{(n)} t^{n-i+1} + t \sum_{i=n+1}^{2n-1} a_i^{(n)} k^{i-n} \quad (5.2.5)$$

a polynomial in  $t$  and  $k$ . Inspection yields-

$$a_1^{(n)} = 1/n! \quad a_{2n-1}^{(n)} = 1/2(n-1)!$$

Exact analysis yields the following for  $n = 2, 3$ , and 4:

$$C(t, 2) = t^2/2 + t/2 + kt/2 \quad (5.2.6)$$

$$C(t, 3) = t^3/6 + 5t^2/4 + 7t/12 + t(k+k^2/4) \quad (5.2.7)$$

$$C(t, 4) = t^4/24 + t^3/3 + 65t^2/24 - t/12 + t(31k/24 + 5k^2/8 + k^3/12) \quad (5.2.8)$$

For an analysis of the two leading terms of the cost function  $C(t, n)$ , we need to know the general form of the second coefficient  $a^{(n)}$ . This is the coefficient corresponding to the coefficient of  $t^n$  in  $C(t, n+1)$ .

Using (2.7) we see that the coefficient of  $t^n$  in

$$\frac{t-1}{n+1} \binom{t+n-1}{n} \text{ is } \frac{n(n-1) - 1}{(n+1)!}$$

Similarly, after some manipulation, we obtain the coefficient of  $t^n$  in

$$\sum_{l=1}^k 2^{l-1} \binom{t/2^{l-1} + n - 1}{n} \text{ as } \frac{1}{2n! (2^{n-1} - 1)}$$

Substituting (5.2.5) into (5.2.4) and rearranging yields the coefficient of  $t^n$  in

$$\sum_{l=1}^k 2^{l-1} C(t/2^{l-1}, n) \text{ as } \frac{2^{n-1}}{n! (2^{n-1} - 1)}$$

Finally, adding and changing  $n+1$  to  $n$ , i.e.,  $a_2^{(n+1)}$  to  $a_2^{(n)}$ , we obtain the coefficient of the second leading term. It is

$$a_2^{(n)} = \frac{1}{2(n-2)!} + \frac{3}{(n-1)! (2^{n-1} - 2)}$$

The coefficient of the leading term, from  $\frac{t-1}{n+1} \binom{t+n-1}{n}$ , is  $a_1^{(n)} = 1/n!$ . Compare the analysis in [2]. We can now compare the leading terms of the cost functions  $B(t,n)$  and  $C(t,n)$  with the lower-limit cost function  $L(t,n)$ . We have:

TABLE 5.3 LEADING TERMS FOR  $B(t,n)$ ,  $C(t,n)$ ,  $L(t,n)$

$n$	$B(t,n)$	$C(t,n)$	$L(t,n)$
2	$5t^2/8$	$t^2/2$	$t^2/2$
3	$t^3/4$	$t^3/6$	$t^3/6$
$n > 3$	$\left[\frac{1}{n!} + \frac{2^{1-n}}{n(n-2)!}\right]t^n$	$t^n/n!$	$t^n/n!$

It is instructive to calculate the relative deviations of  $B(t,n)$  and  $C(t,n)$  from the minimum cost  $L(t,n)$ . We have:

$$[C(t,n) - L(t,n)]/L(t,n) = 3n/(2^{n-1}t) + O(t^{-2})$$

and

$$[B(t,n) - L(t,n)]/L(t,n) = (n-1)t/2^{n-1} + O(1)$$

For large  $t$  and large  $n$ ,  $C(t,n)$  approaches  $L(t,n)$  much more rapidly. In fact, for  $n > 2$  and  $t \geq 16$ , BINC outperforms BINB. (See Appendix I). The fact that there are (at least) four algorithms which are successively better and better than

BINB points up the weakness in the argument that, because  $B(t,n)$  is asymptotically  $L(t,n)$ , therefore BINB may be difficult to improve upon. Asymptotic arguments can be misused. The important thing is to obtain analytically-exact cost functions, and to make a coefficient by coefficient comparison of their leading terms.

### 5.3 ALGORITHM BINS (Binomial S)

We considered an algorithm which is identical to BINC, except that balanced binary splitting was used in place of even splitting. That is, whenever the size of the polynomial to be split was not a power of 2, the polynomial was split as evenly as possible into two polynomials such that the size of one of them was a power of 2. This splitting is less even than even splitting. Hence, it represents a regression with respect to BINC and is, in fact, more expensive. This less attractive algorithm still outperforms BINB for  $t > 12$ . We merely quote the final cost function. Let us suppose that the balanced binary multilevel splitting is  $t = t_1 + \dots + t_p$ ,  $t_i = 2^{k_i}$ . Let  $t^{(0)} = t$ ,  $t^{(i)} = t^{(i-1)} - t_i$ ,  $i = 1, \dots, p-1$ . In this notation  $S(t,n)$  is

$$S(t,n) = \sum_{r=0}^{p-2} S(t^{(r)}, n-1) + \sum_{r=1}^p C(t_r, n) + \sum_{r=1}^{p-1} T_r \quad (5.3.1)$$

where

$$T_r = 2 \binom{t+n-2}{n-1} + \binom{t^{(r-1)}+n-2}{n} - \binom{t^{(r)}+n-2}{n} - \binom{t+n+1}{n}$$

See [2] for more detail.

#### 5.4 ALGORITHM BIND (Binomial D)

This algorithm is specified by the design decisions: binomial expansion, even splitting, multilevel splitting, recursion, and smaller. That is, it is BINC in which left-to-right has been replaced by smaller. More formally:

- (1) Split  $f = f_1 + f_2$ ,  $\text{size}(f_1) = \lceil t/2 \rceil = t_1$   
 $\text{size}(f_2) = \lfloor t/2 \rfloor = t_2$
- (2) For  $r = 2$  to  $n$ , compute  $f_1^r$  and  $f_2^r$  using BIND unless the  $f_i$  is monomial.
- (3) For  $r = 1$  to  $n - 1$ 
  - (a) Multiply  $\binom{n}{r}$  by whichever of  $f_1^r$  and  $f_2^{n-r}$  has fewer terms
  - (b) Multiply this product by the remaining factor.
- (4) Collect the terms of the binomial expansion.

#### Analysis of BIND:

As before,  $D(t, n)$  is the cost in coefficient multiplications of obtaining  $f^n$  when  $\text{size}(f) = t$ . The costs are

itemized as follows:

TABLE 5.4 ITEMIZING THE COSTS IN BIND

Step	Cost
2	$\sum_{r=1}^n [D(t_1, r) + D(t_2, r)]$
3(a)	$\sum_{r=1}^{n-1} \min \left[ \binom{t_1+r-1}{r}, \binom{t_2+n-r-1}{n-r} \right]$
3(b)	$\sum_{r=1}^{n-1} \binom{t_1+r-1}{r} \binom{t_2+n-r-1}{n-r}$

$D(t, n)$  is the sum of expressions in the right-hand column.

We restrict the analysis to  $t = 2^k$ . However,  $D(t, n)$  has been tabulated for  $t \neq 2^k$ . (See Appendix I).

$$D(t, n) = 2 \sum_{r=1}^n D(t/2, r) + 2 \sum_{r=1}^m \binom{t/2+r-1}{r-1} + \delta_n \binom{t/2+n/2-1}{n/2} + \sum_{r=1}^{n-1} \binom{t/2+r-1}{t/2-1} \binom{t/2+n-r-1}{t/2-1} \quad (5.4.1)$$

where  $m = n/2$ ,  $\delta_n = 1$  when  $n$  is even

and  $m = (n-1)/2$ ,  $\delta_n = 0$  when  $n$  is odd.

Using (2.1) and (2.2) this may be rearranged to yield:

$$D(t, n+1) = nt + \sum_{l=1}^k 2^{r-1} D(t/2^{r-1}, n) + \frac{t-1}{n+1} \binom{t+n-1}{n} + \sum_{l=1}^k 2^{r-1} \binom{t/2^r + \lfloor (n-1)/2 \rfloor}{\lfloor (n+1)/2 \rfloor} \quad (5.4.2)$$

If we let  $m = \lfloor (n+1)/2 \rfloor$ , the last term of the right-hand side simplifies to

$$\frac{t}{2m!} \sum_{i=0}^{m-1} \sigma_i \cdot \frac{t^{m-i-1}}{2^{m-i-1}} \quad (5.4.3)$$

where  $\sigma_i$  is the  $i^{\text{th}}$  symmetric function on  $1, 2, \dots, m-1$ . We assume, after inspection of the first few special cases, that the general form for  $D(t, n)$  is

$$D(t, n) = \sum_{i=1}^n a_i t^{n-i+1} + t \sum_{i=n+1}^{2n-1} a_i k^{i-n} \quad (5.4.4)$$

where  $a_i = a_i^{(n)}$ . We have

$$\begin{aligned} \sum_{l=1}^k 2^{r-1} D(t/2^{r-1}, n) &= t \cdot \sum_{j=1}^{n-1} a_j 2^{n-j} \frac{(t^{n-j-1})}{(2^{n-j-1})} + \\ &+ t \cdot \sum_{j=n}^{2n-1} a_j \sum_{s=1}^k s^{j-n} \end{aligned} \quad (5.4.5)$$

Also

$$\begin{aligned} \frac{t-1}{n+1} \binom{t+n-1}{n} &= \frac{t^{n+1}}{(n+1)!} - \frac{t}{n(n+1)} + \frac{t}{(n+1)!} \sum_{j=1}^{n-1} \sigma_j^{(n-1)} - \\ &- \sigma_{j-1}^{(n-1)} t^{n-j} \end{aligned} \quad (5.4.6)$$



Since algorithm BIND, in its leading terms, is the optimal binomial-expansion algorithm which uses recursion for computing powers of subpolynomials, it is worth quoting the closed form of the cost function  $D(t, n)$  in full detail. Substituting the previous four results in (5.4.2) and rearranging gives finally

$$\begin{aligned}
 D(t, n+1) = & \frac{t^{n+1}}{(n+1)!} + t \left[ n - \frac{1}{n(n+1)} - \frac{1}{2m!} \sum_{l=1}^{m-1} \sigma_{r-1}^{(m-1)} / (2^{m-r-1}) - \right. \\
 & - \sum_{j=1}^{n-1} a_j 2^{n-j} / (2^{n-j}-1) \left. \right] + \sum_{n-m+1}^{n-1} t^{n-r+1} \left[ \frac{\sigma_{r+m-n-1}^{(m-1)}}{2m! (2^{n-r}-1)} + \right. \\
 & + \frac{a_r 2^{n-r}}{(2^{n-r}-1)} + \frac{\sigma_r^{(n-1)} - \sigma_{r-1}^{(n-1)}}{(n+1)!} \left. \right] + \sum_{l=1}^{n-m} t^{n-r+1} \left[ \frac{a_r 2^{n-r}}{2^{n-r}-1} + \right. \\
 & + \frac{\sigma_r^{(n-1)} - \sigma_{r-1}^{(n-1)}}{(n+1)!} \left. \right] + kt(a_n + \frac{1}{2}m) + \\
 & + t \sum_{n+1}^{2n-1} a_j \sum_{i=1}^{j-n} i! \binom{j-n}{i} \binom{k+1}{i+1} \quad (5.4.7)
 \end{aligned}$$

This formula gives the coefficients  $a_j^{(n+1)}$  of  $D(t, n+1)$  in terms of the coefficients  $a_j^{(n)}$  of  $D(t, n)$ . Cf. [2].

When  $n = 2, 3, 4$ , we have, always assuming  $t = 2^k$ ,

$$D(t, 2) = t^2/2 + t/2 + kt/2 \quad (5.4.8)$$

$$D(t, 3) = t^3/6 + t^2 + 5t/6 + t(5k/4 + k^2/4) \quad (5.4.9)$$

$$D(t,4) = t^4/24 + 11t^3/36 + 53t^2/24 + 4t/9 + t(7k/4 + 3k^2/4 + k^3/12) \quad (5.4.10)$$

In general

$$D(t,n) = \frac{t^n}{n!} + t^{n-1} \left[ \frac{1}{2(n-2)!} + \frac{1}{(n-1)!(2^{n-2}-1)} \right] + o(t^{n-2}) \quad (5.4.11)$$

The improvement of BIND over BINC is given by

$$C(t,n) - D(t,n) = t^{n-1} \cdot \frac{1}{(n-1)!(2^{n-1}-2)} + o(t^{n-2}) \quad (5.4.12)$$

Another interesting result is that, for  $n \geq 2$ ,

$$\lim_{t \rightarrow \infty} \frac{D(t,n) - L(t,n)}{L(t,n)} = 0 \quad (5.4.13)$$

while

$$\lim_{t \rightarrow \infty} \frac{B(t,n) - L(t,n)}{L(t,n)} = n/2^{n-1} \quad (5.4.14)$$

This last result places BINB in a different asymptotic class from the superior binomial-expansion algorithms developed here. Thus far, we have seen the first few members of a sequence of successively better and better algorithms. With

the exception of the 'S' in BINS, the identifying letters (algorithm names) have all been chosen so that the time-complexity of the named algorithm decreases as we move through the alphabet (BINA, BINB, BINC, BIND, BINE, BINF). There is no BING. As the obvious and then the less obvious and then the subtle improvements are incorporated, it becomes more and more difficult to improve on each successive algorithm. The size of the improvement decreases. BINC differs from BINB in the leading term of the cost function, BIND from BINC in the second leading term. To see clearly the significance of each improvement, we must give exactly the analytic cost function of the new algorithm, and then compare the coefficients of the leading terms against the previous best member of the sequence. After BINC, there are no more improvements in the leading term. BINE, however, is another improvement in the second leading term.

#### 5.5 ALGORITHM BINE (Binomial E)

This algorithm is specified by the design decisions: binomial expansion, even splitting, multilevel splitting, binary merge for computing subpolynomial powers, and smaller. More formally, we make the following description. As usual,

$$f = f_1 + f_2.$$

Description:

- (1) Create the binary term-group tree for the polynomial

f:

- (a) Place f at the root
- (b) Place  $f_1$ ,  $\text{size}(f_1) = \lceil \text{size}(f)/2 \rceil$ , in the left sub-node
- (c) Place  $f_2$  in the right sub-node
- (d) Repeat steps (b) and (c) for the subpolynomials until each term of f has been placed at one of the terminal nodes of the tree

- (2) For each term of the original polynomial f, compute all powers from 2 to n. This completes the processing of the terminal nodes.

- (3) For each strictly interior node, both of whose sub-nodes have already been processed, compute all powers from 2 to n according to the following scheme:

$$(\text{node})^r = (\text{left sub-node} + \text{right sub-node})^r$$

expanded binomially.

- (i) For  $s = 1$  to  $r-1$  do

- (a) Multiply  $(\text{left sub-node})^s$  and  $(\text{right sub-node})^{r-s}$  by whichever of  $(\text{left sub-node})^s$  and  $(\text{right sub-node})^{r-s}$  has fewer terms.

- (b) Multiply the result in (a) by the remaining factor.

- (ii) Collect  $(\text{left sub-node})^r + (\text{right sub-node})^r +$   
the products computed in (i).
- (4) Compute the  $n^{\text{th}}$  power of the root according to the  
previous scheme.

#### Analysis:

The importance of this algorithm justifies giving the analysis in full detail. But for distribution, BINE is (conjecturally) the optimal sequential binomial-expansion algorithm. We will make use of the results and closed-form expressions in Chapter II, and lay the groundwork for the space-complexity analyses in Chapter VI. We begin by developing the concept of a 'power group triangle', which is a representation for the binomial expansions (via sub-nodes) of the powers from 1 to  $n$  of a node, the so-called power group of that node. The representation follows from the 'smaller' idea.

Consider a typical power group, i.e., set of binomial expansions for all powers from 1 to  $n$ , for the  $n$  in  $f^n$ . More precisely, the power group triangle for a node  $a + b$  is the graphical representation in triangular form of all binomial expansions required to compute all powers from 1 to  $n$  of that node, given all powers from 1 to  $n$  of the two sub-nodes.

$$\begin{array}{l}
 a^6 + b^6 + a^5.6b + b^5.6a + a^4.15b^2 + b^4.15a^2 + a^3.20b^3 \\
 a^5 + b^5 + a^4.5b + b^4.5a + a^3.10b^2 + b^3.10a^2 \\
 a^4 + b^4 + a^3.4b + b^3.4a + a^2.6b^2 \\
 a^3 + b^3 + a^2.3b + b^2.3a \\
 a^2 + b^2 + a.2b \\
 a + b
 \end{array}$$

FIG. 5.1 THE POWER GROUP TRIANGLE WHEN  $n = 6$

This is actually a partially-specified algorithm for obtaining all powers from 1 to  $n$  of the polynomial  $a + b$  assuming the availability (apart from binomial coefficients) of all powers from 1 to  $n$  of both subpolynomials. Because  $a$  is the left sub-node, and  $b$  is the right sub-node,  $\text{size}(a) \geq \text{size}(b)$ . The first two columns do not involve computation. In each of the remaining columns, the so-called product columns, care has been taken to group the binomial coefficient together with the smaller of the two polynomials in each cross product. When  $t = 2^k$ , this is rigorously exact. When  $t \neq 2^k$ , there will be a few isolated instances when a larger power of the right sub-node will have fewer terms than a smaller power of the left sub-node. We ignore such instances and maintain the power group triangle in its present form.

Our first analytic task is to evaluate  $BC(s, n)$ , the total binomial-coefficient work required to create the power group of a node whose sub-nodes are of size  $s$ . For instance,

$$BC(s, 2) = s$$

$$BC(s, 3) = 3s$$

$$BC(s, 4) = \frac{1}{2}(11s + s^2)$$

$$BC(s, 5) = \frac{1}{2}(17s + 3s^2)$$

That is, we must evaluate the sum of sizes of polynomials which are multiplied by binomial coefficients.

Now

$$\begin{aligned} BC(s, 2m) - BC(s, 2m-2) &= 4 \sum_{j=1}^{n-1} \binom{s+j-1}{j} + \binom{s+m-1}{m} \\ &= 4 \left[ \binom{s+m-1}{m-1} - 1 \right] + \binom{s+m-1}{m} \quad (5.5.1) \end{aligned}$$

Similarly

$$BC(s, 2m+1) - BC(s, 2m-1) = 4 \left[ \binom{s+m-1}{m-1} - 1 \right] + 3 \binom{s+m-1}{m} \quad (5.5.2)$$

Therefore

$$BC(s, n) - BC(s, n-2) = 4 \left[ \binom{s+p-1}{p-1} - 1 \right] + M_n \binom{s+p-1}{p} \quad (5.5.3)$$

where

$$\begin{aligned} p &= \lfloor n/2 \rfloor \quad \text{and} \quad M_n = 1 \quad \text{when } n \text{ is even} \\ &= 3 \quad \text{when } n \text{ is odd} \end{aligned}$$

Let  $x$  be 2 or 3.  $BC(s, x) = M_n \binom{s+0}{1}$ . A difference scheme, counting down from  $n$  by twos, gives

$$BC(s,n) - BC(s,x) = \sum_{j=1}^{p-1} \{4 \binom{s+j}{j} - 1\} + M_n \binom{s+j}{j+1} \quad (5.5.4)$$

Using (2.1) for the summations and adding  $BC(s,x)$  gives

$$BC(s,n) = 4 \left[ \binom{s+p}{p-1} - p \right] + M_n \left[ \binom{s+p}{p} - 1 \right] \quad (5.5.5)$$

A closed form for  $BC(s,n)$  is therefore:

$$BC(s,n) = \sum_{j=1}^p \left\{ \frac{M_n}{p!} \binom{p+1}{j+1} \right\} + \frac{4}{(p-1)!} \sum_{k=0}^{p-j-1} \binom{p-1}{k+j} \binom{k+j}{k} 2^k s^j \quad (5.5.6)$$

where we have used (2.9) and (2.8), and where it is understood that  $\sum_{k=0}^{-1} \dots$  is zero. The leading term of  $BC(s,n)$  is  $\frac{M_n}{p!} s^p$ .  $BC(s,n)$  is a dense polynomial in  $s$ , of degree  $p$ , minus the constant term; this allows a convenient division of  $BC(s,n)$  by  $s$ . In algorithm BINE, a power group is computed for each strictly interior node, i.e., each node neither terminal nor root. By summing over all interior nodes, we may obtain the total group binomial work, that is, the total number of multiplications by binomial coefficients involved in computing groups of powers. Consider the following term group tree.

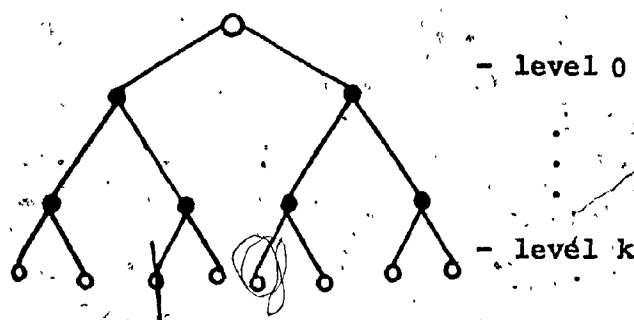


FIG. 5.2 A TERM GROUP TREE



There is no binomial work at level  $k$ , the level of the terminal nodes. There is a power group triangle associated with each of the darkened, strictly interior nodes. The total binomial work here is (counting from the bottom):

$$\frac{t}{2} \cdot BC(1,n) + \frac{t}{4} \cdot BC(2,n) + \dots + 2 \cdot BC\left(\frac{t}{4}, n\right)$$

or

$$\frac{t}{2} \cdot \sum_{j=0}^{k-2} 2^{-j} BC(2^j, n)$$

Now

$$\frac{1}{s} BC(s,n) = \sum_{j=1}^p a_j^{(n)} s^{j-1} \quad (5.5.7)$$

where

$$a_j^{(n)} = \frac{M_n}{p!} \begin{bmatrix} p+1 \\ j+1 \end{bmatrix} + \frac{4}{(p-1)!} \sum_{k=0}^{p-j-1} \begin{bmatrix} p-1 \\ k+j \end{bmatrix} \binom{k+j}{k} 2^k$$

The group binomial work (GBW) is, therefore,

$$GBW = \frac{t}{2} \cdot \sum_{m=0}^{k-2} \sum_{j=1}^p a_j^{(n)} (2^m)^{j-1}$$

$$= \frac{t}{2} \cdot \sum_{j=1}^p a_j^{(n)} \cdot \sum_{m=0}^{k-2} (2^m)^{j-1}$$

$$= \frac{t}{2} \cdot a_1^{(n)} (k-1) + \frac{t}{2} \cdot \sum_{j=1}^{p-1} a_{j+1}^{(n)} \frac{(t/2)^j - 1}{2^j - 1}$$

(5.5.8)

In addition to this group binomial work (GBW), there is root binomial work (RBW).

$$RBW = 2 \sum_{j=1}^{r-1} \binom{t/2+j-1}{j} + N_n \binom{t/2+r-1}{r} \quad (5.5.9)$$

where  $r = \lceil n/2 \rceil$  and  $N_n = 1$  when  $n$  is even  
 $= 0$  when  $n$  is odd.

Hence

$$\begin{aligned} RBW &= 2 \left[ \binom{t/2+r-1}{r-1} - 1 \right] + N_n \binom{t/2+r-1}{r} \\ &= 2 \cdot \text{group}(t/2, r-1) + N_n \cdot \text{size}(t/2, r) \\ &= \frac{2}{(r-1)!} \sum_{j=1}^{r-1} \binom{r}{j+1} \left(\frac{t}{2}\right)^j + \frac{N_n}{r!} \cdot \sum_{j=1}^r \binom{r}{j} \left(\frac{t}{2}\right)^j \quad (5.5.10) \end{aligned}$$

The total binomial work in algorithm binomial E (BINE) is the sum of the group binomial work and the root binomial work. The leading term of that work, when  $p > 1$ , is given by

$$\frac{1}{p!} \left(\frac{t}{2}\right)^p \cdot \left[ 2 - N_n + \frac{M_n}{2^{p-1}-1} \right] \quad (5.5.11)$$

where the first two terms are from RBW and the last term is from GBW. As the leading term of  $L(t, n)$  is  $t^n/n!$ , we see that the leading terms of  $E(t, n)$  result from non-binomial work. That is, since the binomial work does not affect the leading terms of  $E(t, n)$ , the differences in leading terms between  $L(t, n)$  and  $E(t, n)$  are due to non-binomial work.

The total non-binomial work in algorithm BINE is considerably easier to evaluate. It is the sum of the work required to process the terminal nodes and the work required to evaluate all cross products in all binomial expansions as if there were no binomial coefficients in binomial expansions. As before, we will split this non-binomial work into group work and root work. Consider again the term group tree.

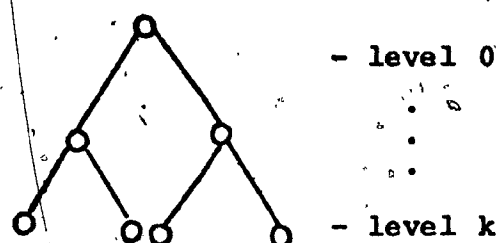


FIG. 5.3 A TERM GROUP TREE

The group work at level  $k$  is  $t(n-1)$ , which is

$$t.\text{group}(1,n) - t$$

If we write the binomial expansion with the binomial coefficients suppressed -

$$f^n = f_1^n + f_2^n + \sum f_1^r f_2^{n-r}$$

-, we see that the non-binomial work to compute  $f^n$  is given by  $\text{size}(f^n) - 2.\text{size}(f_1^n)$ , as there is one coefficient multiplication per extra term formed. Hence, after the group work at level  $k$  has been completed, the additional non-binomial

work required to compute all powers from 2 to n of all nodes at level k-1 is given by  $\frac{t}{2} \cdot \text{group}(2,n) - t \cdot \text{group}(1,n)$ . The total nonbinomial work to level k-1 is the sum

$$\frac{t}{2} \cdot \text{group}(2,n) - t$$

By continuing this argument, the total non-binomial work to level 1, the total group non-binomial work, is just

$$2 \cdot \text{group}(t/2,n) - t$$

The root non-binomial work is simply

$$\text{size}(t,n) - 2 \cdot \text{size}(t/2,n)$$

The total non-binomial work is their sum, namely,

$$\text{size}(t,n) + 2 \cdot \text{group}(t/2,n-1) - t$$

or

$$\frac{1}{n!} \sum_{j=1}^n \binom{n}{j} t^j + \frac{2}{(n-1)!} \sum_{j=1}^{n-1} \binom{n}{j+1} \left(\frac{t}{2}\right)^j - t \quad (5.5.12)$$

which is

$$\frac{t^n}{n!} + t^{n-1} \left[ \frac{1}{2(n-2)!} + \frac{1}{2^{n-2}(n-1)!} \right] + O(t^{n-2}) \quad (5.5.13)$$

This last result is extremely interesting for the following reason. We have seen that the leading terms of  $E(t,n)$  come from the non-binomial work. If the second term in brackets were zero, we would have the two leading terms the same as in

$L(t,n)$ . This means that, even if we reduce the binomial cost to zero, the second term in brackets shows an excess cost (compared to  $L(t,n)$ ) necessarily associated with a binary-merge, binomial-expansion algorithm. We can express that excess cost very simply. Subtracting  $L(t,n)$  from the total non-binomial work gives a non-binomial excess of  $2.\text{group}(t/2, n-1)$ . That is,

$$E(t,n) = L(t,n) + 2.\text{group}(t/2, n-1) + BW,$$

where  $BW$ , the total binomial work, is  $O(t^p)$ ,  $p = \lfloor n/2 \rfloor$ . The complete, closed-form expression for  $E(t,n)$  is the sum of the closed-form expressions for total non-binomial work and total binomial work, namely,

$$\begin{aligned} E(t,n) = & \frac{1}{n!} \sum_{j=1}^n \binom{n}{j} t^j + \frac{2}{(n-1)!} \sum_{j=1}^{n-1} \binom{n}{j+1} \left(\frac{t}{2}\right)^j - t + \\ & + \frac{2}{(r-1)!} \sum_{j=1}^{r-1} \binom{r}{j+1} \left(\frac{t}{2}\right)^j + \frac{N_n}{r!} \sum_{j=1}^r \binom{r}{j} \left(\frac{t}{2}\right)^j + \\ & + \frac{t}{2} \cdot a_1^{(n)}(k-1) + \frac{t}{2} \cdot \sum_{j=1}^{p-1} a_{j+1}^{(n)} \frac{(t/2)^{j-1}}{2^{j-1}} \quad (5.5.14) \end{aligned}$$

where

$$a_j^{(n)} = \frac{M_n}{p!} \binom{p+1}{j+1} + \frac{4}{(p-1)!} \sum_{k=0}^{p-j-1} \binom{p-1}{k+j} \binom{k+j}{k} 2^k$$

$$p = \lfloor n/2 \rfloor \quad N_n = 1, M_n = 1 \quad \text{when } n \text{ is even}$$

$$r = \lceil n/2 \rceil \quad N_n = 0, M_n = 3 \quad \text{when } n \text{ is odd}$$

We may use this to obtain  $E(t,n)$  for  $n = 2, 3, 4$ . We have:

$$E(t,2) = t^2/2 + t/2 + kt/2 \quad (5.5.15)$$

$$E(t,3) = t^3/6 + 3t^2/4 + t/3 + 3kt/2 \quad (5.5.16)$$

$$E(t,4) = t^4/24 + 7t^3/24 + 29t^2/24 - 2t/3 + 11kt/4 \quad (5.5.17)$$

The improvement of BINE over BIND may be expressed as

$$D(t,n) - E(t,n) = t^{n-1} \frac{1}{(n-1)!(2^{n-2}-1)2^{n-2}} + o(t^{n-2}) \quad (5.5.18)$$

In the sequence  $B(t,n)$  to  $E(t,n)$  there is a strictly monotonic decrease both of the cost functions and of the differences between adjacent cost functions. The biggest improvement occurs for the BINB to BINC transition which shows up in the leading term of the cost function. BIND and BINE successively lower the coefficient of the second leading term. After BINE (i.e., BINF) the coefficient of the second leading term is not decreased. It is worth quoting again the two leading terms of  $E(t,n)$ , which, we conjecture, cannot be improved upon. They are not improved upon by BINF, even though BINF is an improvement. These terms are given by

$$E(t,n) = \frac{t^n}{n!} + t^{n-1} \left[ \frac{1}{2(n-2)!} + \frac{1}{2^{n-2}(n-1)!} \right] + o(t^{n-2}) \quad (5.5.19)$$

## 5.6 ALGORITHM BINF (Binomial F)

This algorithm is specified by the design decisions: binomial expansion, even splitting, multilevel splitting, modified merge for sub-polynomial powers, and smaller. BINF differs from BINE only in the handling of multiplication by binomial coefficients near the top of the term group tree. As such, it does not affect the non-binomial work, or the non-binomial-work cost functions, which are responsible for the leading terms of the total time-complexity cost function. The formal description is as follows. As before,  $f = f_1 + f_2$ .

### Description:

- (1) Form the binary term group tree for the polynomial  $f$  in the usual manner: place  $f$  at the root, split  $f$  as evenly as possible placing the slightly larger half (if the sizes are not identical) in the left sub-node and the other half in the right sub-node, and continue this process until monomials are reached.
- (2) Process the terminal nodes (original monomials of  $f$ ) by forming all powers from 2 to  $n$ .
- (3) For all strictly interior nodes other than the two sons of the root, and both of whose sub-nodes have already been processed, compute all powers from 2 to  $n$  by:

$$(\text{node})^r = (\text{left sub-node} + \text{right sub-node})^r$$

expanded binomially.

(i) For  $s = 1$  to  $r-1$  do

(a) Multiply  $\binom{r}{s}$  by whichever of (left sub-node) $^s$  and (right sub-node) $^{r-s}$  has fewer terms.

(b) Multiply the result in (a) by the remaining factor.

(ii) Collect (left sub-node) $^r +$  (right sub-node) $^r +$  the products computed in (i).

(4) For the left and right sub-nodes of the root compute all powers from 2 to  $n$  except for the following:

(a) For the left sub-node, all powers from 2 to  $\lfloor (n-1)/2 \rfloor$ , if any

(b) For the right sub-node, all powers from 2 to  $\lceil (n-1)/2 \rceil$ , if any

(5) Compute the  $n^{\text{th}}$  power of the root according to the following scheme. Use binomial expansion in the manner of (3), that is, form each cross product of the expansion as (larger sub-polynomial power . (binomial coefficient . smaller sub-polynomial power)). If the smaller sub-polynomial has already been computed in (4), compute the inner parenthesis as indicated. Otherwise, compute the inner parenthesis by distributing the binomial coefficient over the summation which forms the binomial expansion of the



smaller sub-polynomial power. That is, if  $g^r$  is the smaller sub-polynomial power, compute  $\binom{n}{r} g^r$  as

$$\binom{n}{r} g_1^r + \binom{n}{r} g_2^r + \sum_s \binom{n}{r} \binom{r}{s} g_1^s g_2^{r-s}$$

where

$$f = g + h \text{ and } g = g_1 + g_2$$

### Analysis:

We are interested in evaluating  $E(t,n) - F(t,n)$ , the number of binomial coefficient multiplications, if any, saved by not computing all powers from 2 to  $n$  of the left and right sub-nodes of the root. This savings of binomial coefficient multiplications accounts for all of the difference in time-complexity between BINE and BINF. The use of distribution to by-pass the independent computation of subpolynomial powers effectively reduces the amount of root binomial work; the most direct way to compute  $F(t,n)$ , then, is simply to re-evaluate the function  $RBW(t,n)$ . We have seen previously that, in BINE, this function is given by

$$\begin{aligned} RBW &= 2 \left[ \binom{t/2+r-1}{r-1} - 1 \right] + N_n \binom{t/2+r-1}{r} \\ &= 2 \cdot \text{group}(t/2, r-1) + N_n \cdot \text{size}(t/2, r) \quad (5.6.1) \end{aligned}$$

where

$$\begin{aligned} r &= \lceil n/2 \rceil \text{ and } N_n = 1 \text{ when } n \text{ is even} \\ &= 0 \text{ when } n \text{ is odd} \end{aligned}$$

In BINF, distribution is used to evaluate products of the form  $\binom{n}{s} g^s$  in precisely two cases: (a) when  $g$  is the left sub-node of the root, and  $s$  lies between 2 and  $\lfloor (n-1)/2 \rfloor$ , and (b) when  $g$  is the right sub-node of the root, and  $s$  lies between 2 and  $\lceil (n-1)/2 \rceil$ . If  $s = g_1 + g_2$ , then distribution means: compute  $\binom{n}{s} g^s$  as

$$\binom{n}{s} g_1^s + \binom{n}{s} g_2^s + \sum_{j=1}^{s-1} \binom{n}{s} \binom{s}{j} g_1^j g_2^{s-j}$$

That is, if we allow for the cost of computing the  $\binom{n}{s} \binom{s}{j}$ , we avoid the cost of multiplying  $\binom{n}{s}$  by the product terms in the binomial expansion of  $g^s$ . The cost of forming the  $\binom{n}{s} \binom{s}{j}$  is  $\lfloor s/2 \rfloor$ , the number of distinct  $\binom{s}{j}$  when  $1 \leq j \leq s-1$ ; the resultant savings is that the binomial cost of computing  $\binom{n}{s} g^s$  drops from  $\text{size}(g^s)$  to  $2 \cdot \text{size}(g_1^s)$ . (The cost of multiplying the  $\binom{n}{s} \binom{s}{j}$  by the appropriate polynomials is already accounted for in the binomial cost of computing  $g^s$ , i.e., in GBW.) We note that, when  $s = 1$ , there are no product terms in the binomial expansion of  $g^s$ . Thus where, for BINE, RBW was given by

$$\text{RBW} = 2 \cdot \sum_{j=1}^{r-1} \binom{t/2+j-1}{j} + N_n \cdot \binom{t/2+r-1}{r} \quad (5.6.2)$$

it is now, for BINF, given by

$t = 2^k$ . Construct the sequences  $S_i$ ,  $1 \leq i \leq t$ , whose  $j^{\text{th}}$  term,  $1 \leq j \leq \text{size}(t, r-1)$ , is obtained by multiplying the  $i^{\text{th}}$  term of  $f$  by the  $j^{\text{th}}$  term of  $f^{r-1}$ . That is,  $S_i$  is the product of  $f^{r-1}$  with the  $i^{\text{th}}$  term of  $f$ . Next, merge  $S_{2i-1}$  with  $S_{2i}$  for  $1 \leq i \leq t/2$ , combining terms. Then merge the resulting sequences in pairs, combining terms, until one sorted sequence remains. In theory, the sorting time, namely,

$$O(t.k.\text{size}(t, r-1))$$

dominates the multiplication cost, namely,  $t.\text{size}(t, r-1)$ , but we are ignoring sorting time both for the sake of uniformity and to be generous to both RMUL and hence BINB. (Fateman asserts that the sorting time for any practical problem appears to be negligible [4]). None of our algorithms requires any sorting whatsoever, when the polynomials are completely sparse.

The largest core requirements for computing  $f^n$  by repeated multiplication occur during the step  $f^n = f.f^{n-1}$ . If all of the subsequences are obtained before merging, then  $t.\text{size}(t, n-1)$  terms need to be stored. This is a merge sort of  $t.\text{size}(t, n-1)$  numbers with  $t$  runs. Alternatively, if each subsequence is merged immediately after its formation, we are interested in the space required to merge the last subsequence with the sequence which is the union (by merge sort) of all previous subsequences. A merge sort of  $n$  records typically requires  $2n$  locations. We can avoid this, and sort in place, by applying a list merge sort (rearranging pointers) to polynomials represented as linked lists. The combined size

of the last two lists to be merged does not greatly exceed  $\text{size}(t,n)$ , the size of the final result. Given the diversity of multiplication-plus-sorting schemes, it is simplest to assign the following space complexity to RMUL, which is, in fact, an extremely generous lower limit. Taking the combined list size at the end as roughly  $\text{size}(t,n)$ , we assign a space complexity of  $\text{size}(t,n)$ , provided a link field is attached to each term. In that case, if  $\text{size}(f) = t$ , the space complexity of computing  $f^n$  by repeated multiplication is given by  $\text{size}(t,n) \cdot (1+E+P)$  central memory words.

It is our general conclusion that a linked-list representation for polynomials, with a storage requirement of  $(1+E+P)$  central-memory words per polynomial term, makes good sense from both the time-complexity and space-complexity stand-points, and this for all the sequential binomial-expansion algorithms considered in this thesis. If this be so, we need only measure the space complexity as number of terms of storage required. The final answer,  $f^n$ , may reasonably be written to disc, or some other form of secondary storage. Our real concern, then, is to determine, for each algorithm, the indispensable minimum core-storage required by the computation: This working storage will be essentially the space to store intermediate results after they have been obtained but before they have been used. The different ways in which the various algorithms generate and use intermediate results naturally give rise to different space complexities. As binary merge is a streamlined form of recursion (very much in the spirit, actually, of dynamic programming) in which pre-

cisely the minimum amount of intermediate results is generated, we concentrate here on comparing the space complexity of the algorithms which use merge for subpolynomial powering with the space complexity of the algorithms which use repeated multiplication. A family of implementation strategies for BINE will be considered, leading ultimately to a new algorithm, BINF.

#### 6.1 SPACE ANALYSIS FOR BINA AND BINB (Binomial A and Binomial B)

We consider in-core, linked-list implementations of these two algorithms, where the final answer,  $f^n$ , may be written to disc, and calculate the minimum storage required to store the intermediate results. The space complexities of BINA and BINB have never been analysed; in the case of BINB, we are faced with decisions concerning the implementation strategy which will radically affect the space complexity of the algorithm. In these algorithms,  $f^n$  is computed as

$$f_1^n + f_2^n + \sum_{r=1}^{n-1} \binom{n}{r} f_1^r f_2^{n-r}$$

where the powers of  $f_1$  and  $f_2$  are first computed by repeated multiplication. The difference is that, in BINA,  $\text{size}(f_1) = 1$  and  $\text{size}(f_2) = t-1$ , while in BINB,  $\text{size}(f_1) = \text{size}(f_2) = t/2$ . In the first algorithm, i.e., BINA, the space required to compute  $f_2^n$  dominates all other storage requirements. Using the previous generous lower limit for the space complexity of RMUL, we may say that the space complexity of algorithm BINA is  $\text{size}(t-1, n)$  terms. This may also be written

$$\begin{aligned}
 s_A &= \frac{1}{n!} \sum_{j=1}^n \binom{n}{j} (t-1)^j \\
 &= \frac{t^n}{n!} + \frac{1}{n!} \sum_{j=1}^{n-1} (\binom{n-1}{j-1} - \binom{n-1}{j}) t^j \quad (6.1.1)
 \end{aligned}$$

The coefficient of  $t^{n-1}$  is

$$\frac{1}{n!} \left[ \binom{n-1}{2} - 1 \right]$$

this space complexity is not radically less than  $\text{size}(t, n)$ , the size of the final answer.

That the space complexity of BINA is  $\text{size}(t-1, n)$  may be seen as follows. Because the sizes of the  $f_1^r$  are insignificant, one need not worry which subpolynomial powers to keep in core, nor in what order; one first generates all the  $\binom{n}{r} f_1^r$ , stores them, and then generates the successive powers of  $f_2$  (from 2 to  $n$ ) using each power in the binomial expansion as soon as it has been generated. In a word, the tasks of computing the  $\binom{n}{r} f_1^r f_2^{n-r}$  are independent subtasks. The most space is required to generate  $f_2^n$ , as explained. Yet no two powers of  $f_2$  (apart from  $f_2$  itself, of course) need be present in core at the same time. (One is assuming sophisticated garbage collection here, and the ability to overwrite  $f^n$  in the cells which held  $f^{n-1}$ .) Thus, ideally, one can manage with only  $\text{size}(t-1, n)$  cells, since  $O(t)$  cells more is negligible. In BINB, on the other hand, the sizes of  $f_1$  and  $f_2$ , and of their respective powers, are more or less the same. One must choose which powers to store in core, and this is not an easy choice.

If one attempts to hold all the powers of both  $f_1$  and  $f_2$  in core before expanding binomially, the total space required is  $2 \cdot \text{group}(t/2, n)$ , that is, twice the sum of sizes of all powers from 1 to  $n$  of a polynomial of size  $t/2$ . This is

$$\frac{2}{n!} \sum_{j=1}^n \binom{n+1}{j+1} \left(\frac{t}{2}\right)^j \quad (6.1.2)$$

which is not small, but much less than  $\text{size}(t, n)$ .

Two times  $\text{group}(t/2, n)$  is not a lower bound for the space complexity of BINB: we can do better. We observe first that a power of  $f_1$  or  $f_2$  not needed to generate higher powers, and already used in the binomial expansion of  $f^n$ , need not be retained in core. The non-negligible sizes of both the  $f_1^r$  and the  $f_2^r$  give us less flexibility here, but the following approach may be tried. (We maintain the, perhaps overgenerous, assumption that  $f_1^r = f_1 \cdot f_1^{r-1}$  may be computed in space  $\text{size}(t/2, r)$ .) First, generate all powers from 2 to  $p$  of both  $f_1$  and  $f_2$ , where  $p = \lfloor n/2 \rfloor$ . If  $n$  is even, the product  $\binom{n}{p} f_1^p f_2^p$  may now be formed, and  $p^{\text{th}}$  powers are no longer required in the binomial expansion. Next, successively generate the powers from  $p+1$  to  $n$  of  $f_1$ , using each power in the expansion as soon as generated (one uses  $f_1^n$  by writing it to disc), releasing powers of  $f_2$  whenever possible (once they have been used), and not retaining powers of  $f_1$  beyond  $f_1^p$ . Finally, successively generate the powers from  $p+1$  to  $n$  of  $f_2$ , proceeding in exactly the same manner. The least value of the minimum space required

occurs when  $n$  is even. After forming  $\binom{n}{p} f_1^p f_2^p$  one needs to retain the powers from 1 to  $p-1$  of  $f_1$  (to match the as yet ungenerated higher powers of  $f_2$ ), and  $f_2^p$  (to generate these higher powers.) One also requires space to generate  $f_1^n$ . Thus, at best, the space complexity of algorithm BINB is  $\text{size}(t/2, n) + \text{group}(t/2, p)$ . This may also be written

$$S_B = \frac{1}{n!} \sum_{j=1}^n \binom{n}{j} \left(\frac{t}{2}\right)^j + \frac{1}{p!} \sum_{j=1}^p \binom{p+1}{j+1} \left(\frac{t}{2}\right)^j,$$

$$p = \lfloor n/2 \rfloor \quad (6.1.3)$$

For future reference, we list the leading terms corresponding to  $\text{size}(t, n)$  and to the space complexity functions obtained so far.

$$\text{size}(t, n) = \frac{t^n}{n!} + o(t^{n-1}) \quad (6.1.4)$$

$$S_A = \frac{t^n}{n!} + o(t^{n-1}) \quad (6.1.5)$$

$$2 \cdot \text{group}(t/2, n) = \frac{t^n}{n! \cdot 2^{n-1}} + o(t^{n-1}) \quad (6.1.6)$$

$$S_B = \frac{t^n}{n! 2^n} + o(t^{n-1}) \quad (6.1.7)$$

These functions are listed in strictly decreasing order. One sees that algorithm BINB has an impressively low space complexity, essentially that of the size of its single largest intermediate result,  $f_1^n$ . (The rest is asymptotically negligible.) To attempt to match the BINB space complexity, a family of



implementation strategies for BINE will be considered, with the same time complexity but better and better space complexities, until finally BINF is considered, needing less time and requiring significantly less space. BINF was developed for space reasons. Of all known sequential algorithms for this problem, BINF has the least time and the least space.

## 6.2 SPACE ANALYSIS FOR BINE AND BINF (Binomial E and Binomial F)

We begin by establishing a result which we have tacitly assumed up until now, namely, if  $f_1^r$  and  $f_2^{n-r}$  are present in core then  $\binom{n}{r} f_1^r f_2^{n-r}$  may be obtained with space complexity not exceeding the space, if any, required to store the result. (That is, there is no need for working storage.) We form the product by retrieving each term of the smaller polynomial, multiplying by the binomial coefficient, and then retrieving and multiplying by each term of the larger polynomial, from which the result follows. Next, we consider the space required to store all powers from 1 to  $n$  of all subpolynomials associated with nodes of the multilevel term-group tree; this is potentially a rather large number. Consider a father node,  $f$ , and the two subnodes,  $f_1$  and  $f_2$ . Suppose  $f_1^n$  and  $f_2^n$  are present in core. We compute  $f^n$  according to

$$f^n = f_1^n + f_2^n + \sum \binom{n}{r} f_1^r f_2^{n-r}$$

The additional space required to store  $f^n$  is the space for the terms from the cross products. Suppose now the groups of

powers of all nodes at level  $k$  are present in core. The additional space required to store the groups of powers of all nodes at level  $k-1$  is the space for the terms from the cross products. Continuing this argument, we may form all powers from 1 to  $n$  of all nodes at all levels from  $k$  to 1 (level 1 corresponds to the two subnodes of the root) until finally our space requirements increase to  $2.\text{group}(t/2, n)$ .

The linked-list representation for polynomials allows us to hold all of level  $k$  in core, and then all of level  $k-1$ , and so on, up to level 1, the level of the two sub-nodes of the root, without any garbage collection. This is because any power of any node at level  $j$  belongs to the binomial expansion of the same power of some node at level  $j-1$ . For example, if

$$g^r = g_1^r + g_2^r + \binom{r}{s} g_1^s g_2^{r-s}$$

the linked list for  $g^r$  contains, first, the terms from the cross products, and second, the two previously existing lists,  $g_1^r$  and  $g_2^r$ . List concatenation absorbs the lists at a level into the lists at the next higher level. As the levels of the term-group tree are successively constructed (starting from the terminal nodes), no space is ever released; rather, with each new level, more space must be allocated for the new terms required to form that level. The most space required is that for the highest level formed, here,  $2.\text{group}(t/2, n)$ . Thus, a naive approach to implementing BINE yields exactly the same space complexity as the naive approach to implementing BINB,

except here there is no generous lower limit, sophisticated sorting, overwriting and garbage collection, just straight computation. Dynamic programming is a kind of recursion in which one keeps track of subproblems and never solves the same problem twice; the term-group tree, with its groups of powers of subpolynomials, is the table of solutions to all subproblems. The simplicity of the space complexity comes about because the solutions to the smaller subproblems are part of the solutions to the larger subproblems.

We have agreed that the final answer,  $f^n$ , may be written to disc rather than retained in core. Thus, once the groups of powers of the root subnodes have been obtained, there is no need for additional central memory. By retaining a full power group for each node other than the root, and writing the  $n^{\text{th}}$  power of the root out onto disc, we require  $2 \cdot \text{group}(t/2, n)$  terms of storage. Yet there is no need ever to retain the  $n^{\text{th}}$  power of any subpolynomial. Let the two subnodes of a node  $g$  be  $g_1$  and  $g_2$ .

$$g^n = g_1^n + g_2^n + \sum \binom{n}{r} g_1^r g_2^{n-r}$$

Suppose that  $g_1^n$  and  $g_2^n$  have been written to disc, and that all other powers of  $g_1$  and  $g_2$  are available in core. By writing the requisite cross products  $\binom{n}{r} g_1^r g_2^{n-r}$  to disc, we have written  $g^n$  to disc. We may in fact, write to disc the  $n^{\text{th}}$  power of every node, including in core only the 'limited' power group for each node, i.e., always excluding

power  $n$ . Not storing the  $n^{\text{th}}$  powers at level 1 (an obvious first thought) gives BINE a space complexity of  $2.\text{group}(t/2, n-1) + 4.\text{size}(t/4, n)$ ; not storing the  $n^{\text{th}}$  powers at any level gives BINE a much better space complexity, namely,  $2.\text{group}(t/2, n-1)$ . This may also be written

$$S_E = \frac{2}{(n-1)!} \sum_{j=1}^{n-1} \left[ \begin{matrix} n \\ j+1 \end{matrix} \right] \left( \frac{t}{2} \right)^j \quad (6.2.1)$$

The leading term here is  $\frac{t^{n-1}}{(n-1)! 2^{n-2}}$ , and  $S_E < S_B$ , asymptotically.

So far we have discussed two sophisticated implementations of BINE, both with asymptotically smaller space complexities than the lower limit for the space complexity of BINB, both based on the idea of not retaining  $n^{\text{th}}$  powers of subpolynomials in core. BINB can make no use of this idea; in contrast to recursion or dynamic programming (here, binary merge), repeated multiplication is a whole polynomial method committed to building up  $f_1^n$  and  $f_2^n$  in core. To see the relative magnitudes of the space complexity functions for BINB (lower limit) and BINE (two implementations), consider the following three expressions.

$$S_B = \frac{1}{n!} \sum_{j=1}^n \left[ \begin{matrix} n \\ j \end{matrix} \right] \left( \frac{t}{2} \right)^j + \frac{1}{p!} \sum_{j=1}^p \left[ \begin{matrix} p+1 \\ j+1 \end{matrix} \right] \left( \frac{t}{2} \right)^j, \quad p = \lfloor n/2 \rfloor \quad (6.2.2)$$

$$2.\text{group}(t/2, n-1) + 4.\text{size}(t/4, n) = \frac{2}{(n-1)!} \sum_{j=1}^{n-1} \binom{n}{j+1} \left(\frac{t}{2}\right)^j + \frac{4}{n!} \sum_{j=1}^n \binom{n}{j} \left(\frac{t}{4}\right)^j \quad (6.2.3)$$

$$S_E = \frac{2}{(n-1)!} \sum_{j=1}^{n-1} \binom{n}{j+1} \left(\frac{t}{2}\right)^j \quad (6.2.4)$$

These three expressions are listed in strictly decreasing asymptotic order. The first has leading term  $\frac{t^n}{n! 2^n}$ ; the third has leading term

$$\frac{t^{n-1}}{(n-1)! 2^{n-2}}$$

When  $t$  grows large, with  $n$  fixed,  $S_B$  grows faster than  $S_E$ . (The cross-over point for the leading terms occurs for  $t = 4n$ .) Still, there will be values of  $t$  and  $n$  for which the BINB lower limit is less than the BINE actual value. We need to improve the space complexity of our use of dynamic programming.

When discussing the lower limit for BINB space complexity, we saw that the tasks of computing the  $\binom{n}{r} f_1^r f_2^{n-r}$  are essentially independent subtasks. There is no particular reason, when performing one subtask, to store the intermediate results necessary to perform some other subtask. As always,

$$f^n = f_1^n + f_2^n + \sum_{r=1}^{n-1} \binom{n}{r} f_1^r f_2^{n-r}$$

where  $f$  is the root, and  $f_1$  and  $f_2$  the nodes at level 1. Both  $f_1^n$  and  $f_2^n$  have already been written to disc. We now take the decision to generate the powers of nodes at level 1 (assuming the existence of the powers of nodes at level 2) only as they are successively needed in the binomial expansion. When  $t$  is sufficiently large, the most space required will be that to generate  $\binom{n}{1} f_1^{n-1} f_2$  (or vice-versa). Taking into account the groups of powers at level 2, we obtain a space complexity  $\underline{S}_E$  of size  $(t/2, n-1) + 4 \cdot \text{group}(t/4, n-1)$ , using a bar to distinguish the new space complexity function. We may write this BINE space complexity as

$$\begin{aligned} \underline{S}_E = & \frac{1}{(n-1)!} \sum_{j=1}^{n-1} \binom{n-1}{j} \left(\frac{t}{2}\right)^j + \\ & + \frac{4}{(n-1)!} \sum_{j=1}^{n-1} \binom{n}{j+1} \left(\frac{t}{4}\right)^j \end{aligned} \quad (6.2.5)$$

Surely,  $\underline{S}_E = \text{size}(t/2, n-1) + 4 \cdot \text{group}(t/4, n-1)$ , the lowest obtainable space complexity for algorithm BINE, is less than, i.e., asymptotically less than

$$S_B = \text{size}(t/2, n) + \text{group}(t/2, p), p = \lfloor n/2 \rfloor$$

which is a lower bound on the space complexity of algorithm BINB. The first term of the sum dominates in both cases, and  $\text{size}(t/2, n) \gg \text{size}(t/2, n-1)$ . The leading term of  $\underline{S}_E$  is

$$\frac{t^{n-1}}{(n-1)!} \left[ \frac{1}{2^{n-1}} + \frac{1}{2^{2n-4}} \right]$$

It seems extremely difficult to avoid in-core storage for the powers from 1 to  $n-1$  of the nodes at level 2; these powers are used again and again at level 1: they should be computed once, and stored. An improvement at level 1 with respect to storage is, however, possible, if we make use of distribution to precompute directly products of the form  $\binom{n}{r} f_1^r$ , which is the characterizing idea of algorithm BINF. Consider writing (say)  $f^4$  as

$$f_1^4 + f_1^3 \cdot 4f_2 + f_2^3 \cdot 4f_1 + f_1^2 \cdot 6f_2^2,$$

i.e., according to the smaller idea. Apart from  $f_1^4$  and  $f_2^4$ , which have already been written to disc, the binomial expansion of the desired power of the root consists of a number of cross products of the form, larger polynomial times binomial coefficient times smaller polynomial. The larger polynomial will be called the a-list; the product of the smaller polynomial by the binomial coefficient will be called the b-list. The sum of interest thus becomes

$$\sum_i \text{a-list}_i \cdot \text{b-list}_i$$

These new objects, namely, a-lists and b-lists, belong to level 1 of the term-group tree. We suppose that level 2 has already been processed, and that all powers from 1 to  $n-1$

of all sub-polynomials at level 2 are available in core. The claim is made that the  $n^{\text{th}}$  power of the root may be written to disk using only the additional amount of central memory required to store the largest b-list. This gives algorithm BINF a total space complexity of  $\text{size}(t/2, p) + 4 \cdot \text{group}(t/4, n-1)$ ,  $p = \lfloor n/2 \rfloor$ . This may also be written

$$S_F = \frac{1}{p!} \sum_{j=1}^p \binom{p}{j} \left(\frac{t}{2}\right)^j + \frac{4}{(n-1)!} \sum_{j=1}^{n-1} \binom{n}{j+1} \left(\frac{t}{4}\right)^j,$$

$$p = \lfloor n/2 \rfloor \quad (6.2:6)$$

The leading term of  $S_F$  is  $\frac{t^{n-1}}{(n-1)! 2^{2n-4}}$ . Comparing the leading term of  $S_B$ , viz.,

$$\frac{t^n}{n! 2^n}$$

we see that  $S_F$  is far superior. The claim made above may be substantiated as follows. Consider a-list<sub>i</sub>, b-list<sub>i</sub> for a particular value of  $i$ . By distribution of the binomial coefficient over the binomial expansion of the smaller polynomial, the b-list may be represented in terms of constants and powers of sub-polynomials available at level 2. Hence, it may be computed, and stored in core in a space able, by definition, to hold the largest b-list. That space is of size  $\text{size}(t/2, p)$ .

We continue the argument. One particular b-list is now available in core. The corresponding a-list possesses a



binomial expansion in terms of binomial coefficients and powers of subpolynomials available at level 2. This a-list may be computed at essentially no additional central memory cost. Let the a-list be  $g^r$ , and let the binomial expansion be

$$g^r = g_1^r + g_2^r + \sum \binom{r}{s} g_1^s g_2^{r-s}$$

For each term of  $g_1^r$  and  $g_2^r$ , multiply by each term of the b-list, and write this product to disc. For the cross products, a slightly different strategy is adopted. Each cross product has a larger polynomial and a smaller polynomial, as usual. For each term of the smaller polynomial, multiply by the binomial coefficient, then by each term of the larger polynomial, and finally by each term of the b-list. The products computed in the inner loop are written to disc. In pseudo-Pascal, this is:

```

for each term of small do
  temp := coefficient * small [i]
  for each term of large do
    for each term of b-list do
      write(temp*large [j]*b-list [k])

```

The whole loop, of course, is executed once for each cross product in the binomial expansion of the a-list in question.

We can catalogue the various space-complexity functions obtained so far. The lower bound on the BINB space complexity is  $\text{size}(t/2, n) + \text{group}(t/2, p)$ ,  $p = \lfloor n/2 \rfloor$ . The space complex-

ities of the various implementations of BINE are:  $2.\text{group}(t/2, n)$ ,  $2.\text{group}(t/2, n-1) + 4.\text{size}(t/4, n)$ ,  $2.\text{group}(t/2, n-1)$ , and  $\text{size}(t/2, n-1) + 4.\text{group}(t/4, n-1)$ . The space complexity of the essentially unique implementation of BINP is  $\text{size}(t/2, p) + 4.\text{group}(t/4, n-1)$ ,  $p = \lfloor n/2 \rfloor$ . The E and F complexities do represent the space that would be used by these implementations; the B complexity is a lower bound which is rather generous, especially for small values of  $n$  and  $t$ . Clearly, we want to compare the relative values of these functions. The E and F series, as written above, is strictly monotonically decreasing; each function is obtained from the previous by subtracting a strictly positive quantity. The claim was made above that the E and F implementations which did not retain  $n^{\text{th}}$  powers in core were asymptotically superior to B, essentially on the grounds that  $n^{\text{th}}$  powers of sub-polynomials eventually outgrow  $n-1^{\text{st}}$  powers. But there is a danger in all such asymptotic arguments that the asymptotically superior algorithm becomes superior just as we pass beyond the bounds of the practically computable. Hence, we shall now make a more careful comparison of  $S_B$  and  $S_F$ , the space complexities, respectively, of the most successful repeated-multiplication algorithm, and the most successful dynamic-programming algorithm.

When  $t = 2$ , not too surprisingly, the differences between binomial-expansion algorithms disappear. Here, there is no difference between single-level and multi-level, dynamic programming is repeated multiplication, and no power of a monomial is smaller than any other. All the formulas given so far for time

complexity and space complexity, with one exception, were derived under the implicit assumption that  $t$  was a power of two; when this is not the case, the formulas are only approximate. To see the relative magnitudes of  $S_B$  and  $S_F$  very clearly, these functions were tabulated for various values of  $t = 2^k$  and  $n$ . When  $t = 4$ , BINB requires less space; the storage for intermediate results, however, is less even than the space required to store the program. When  $t \geq 8$ , BINF requires less space; in the one case of equality ( $t = 8$ ,  $n = 3$ ), the BINB lower limit can be shown to be inapplicable. Examination of the tabulated results shows that, in fact, it is over the whole range of the practically computable (apart from the trivially small) that the space complexity of BINF is superior to that of BINB. For a small problem,  $t = 8$ ,  $n = 10$ ,  $S_B = 411$ ,  $S_F = 272$ ; for a large problem  $t = 32$ ,  $n = 5$ ,  $S_B = 15,656$ ,  $S_F = 2,112$ . One may conclude, therefore, that dynamic programming, coupled with intelligent memory management, leads to space improvements more dramatic even than those in the time domain. One is certainly very far from the classical idea of a space-time trade-off. (The Table is Appendix III.)

This concludes the discussion of the sequential binomial-expansion algorithms. The results obtained depend totally on the two models for the problem which have been adopted in this thesis, namely, the cost model and the computational model. Nothing extends the validity of these results to other models. The computational model has been that the input polynomials be multivariate polynomials completely or almost completely sparse

to power  $n$ ,  $n$  the power sought. Obviously not all polynomials are sparse, yet existing algebra systems often need to handle multivariate problems of a sparse nature. The algorithms treat almost sparse polynomials as if they were totally sparse. The cost model has been that the true run-time cost may accurately be measured by the number of coefficient multiplications used in the algorithm to generate the final result; this model has been justified above. Finally, the space complexity of an algorithm has been defined as the central memory required to store intermediate results in the best implementation of that algorithm; assuming a linked-list representation for polynomials, space complexities have been quoted in number of terms of storage required. A term may require several central memory words, the exact number of which is algorithm-independent. These, then, are the chief assumptions about the input polynomials, and about the ways to measure the time and space complexities of the algorithms.

Very definite conclusions have been reached about which algorithms have least time complexity, and which least space complexity (BINF, apparently, is optimal in both respects.) In addition, conclusions have been drawn about the desirability of various design options which exist in sequential binomial-expansion algorithms. These conclusions, too, are model-dependent. One example can be given. When the polynomials are completely sparse, clearly it is best to split the polynomials as evenly as possible. This reduces costs, as we have

seen. Yet nothing rules out the possibility that, for completely dense polynomials, the best choice would have been to split the polynomials as unevenly as possible. And so on for the other decisions. The final conclusion is this. In this section we have analysed a family of sequential binomial-expansion algorithms for symbolic computation of integer powers of completely or almost completely sparse polynomials. We have analysed the time and space complexities of these algorithms. By a series of refinements and improvements, based on the ideas of dynamic programming and intelligent memory management, we have arrived at an algorithm, algorithm BINP (pronounced: binomial P), which we believe and conjecture to be optimal for both time and space within the binomial-expansion family under the assumptions listed above. Possibly, it is the optimal way to power sparse polynomials.

CHAPTER VII

PARALLEL ALGORITHMS

## CHAPTER VII

### PARALLEL ALGORITHMS

In the discussion of the sequential algorithms we saw that the theoretical lower limit on the number of coefficient multiplications required to compute the  $n^{\text{th}}$  power of an arbitrary polynomial was given by  $\text{size}(t, n) - t$ , which, for large  $t$  and large  $n$ , can become an extremely large number. This result puts limits on what is practically computable with a sequential architecture. Recent technological advances, in principle, make possible the production of very cost-effective high-performance computers which make parallel use of a large number of processors. Researchers have analysed various parallel computer architectures, and the problems of adapting sequential algorithms to parallel machines, in applications areas where enormous amounts of straight computation are required. A major lesson has been that special-purpose machines which have been adequately tailored to their applications area can perform spectacularly, even if these same machines do much less well when applied to problems they were never intended to solve. We consider two special-purpose parallel architectures, one multiprocessor and the other associative-processor, then examine the adaptations necessary to run variants of the best sequential binomial-expansion algorithms on these machines, and finally calculate the speed-up ratio obtained for each architecture-algorithm combination. The general conclusion is that integer powers of sparse

polynomials are well-suited to parallel computation, with the actual speed-up ratios approaching the theoretical ideal.

A basic division of parallel architectures [ 6 ] is into single instruction stream, multiple data stream systems, and multiple instruction stream, multiple data stream systems. (There are other types.) In the first category, only one instruction is executing at any one time (single control unit), yet may act on a whole set of data (multiple processing units); examples are array processors and associative processors. In the second category many instructions execute simultaneously (multiple control units) on different data (associated multiple processing units); essentially, we have a system of interconnected conventional processors. Of course, an architecture can be devised which lies somewhere between these two extremes. (The two extremes are having many control-less functional units, and having many general-purpose computers.) In the associative-processor architecture envisaged here, a large associative memory (in which data is addressed by tag rather than by address) will be coupled with a parallel processing array whose elements perform more or less the same computation simultaneously. In the multiprocessor architecture, there will be a central control section, capable of sophisticated processing, and a number of slave processors with extremely limited control capabilities. That is, the former system will essentially be SIMD (single instruction stream, multiple data stream,) while the other system will keep the deviations from the SIMD concept to an acceptable minimum.



In general, one cannot obtain good parallel algorithms by simple translation of existing serial algorithms. This is because a particular parallel architecture will be efficient only if the computation to be run on it has a particular form. If one is running on a SIMD machine, then the original computation must be broken up into many smaller subcomputations which are structurally identical but which may have different data values. When such a splitting is not possible, one is forced to use an MIMD machine, in which the subcomputations have different structures as well as different data values. Thus, in the parallel case, on the one hand, there is a definite algorithm-machine interdependence, and, on the other hand, starting with a given serial algorithm for parallel adaptation will place constraints on the way the computation may be broken up into subcomputations. In the sequential binomial-expansion algorithms, one multiplies constants times polynomials, and polynomials times other polynomials. The first operation is monomial times polynomial; the second operation is monomial times polynomial many times. The most elementary operation is simply monomial times monomial. One way to achieve parallelism, insofar as the elementary operations are completely independent, is to split a composite task, such as polynomial times polynomial, into sets of elementary sub-tasks. Another way to achieve parallelism, specific to algorithms which employ multi-level splitting, is to take advantage of the structurally-identical subcomputations associated with different nodes belonging to one particular level of the term-group tree. These

are the two main approaches. Both are used in connection with the multiprocessor architecture; the former alone is used in connection with the associative-processor architecture.

One needs a criterion to measure the success of a particular parallel algorithm-architecture combination, that is, essentially, some measure of whether the decrease in run-time warrants the additional expenditure for the parallel machine. Following Stone [14], we adopt the measure for specific problems of the speed-up ratio defined as:

$$\text{Speed-up ratio} = \frac{\text{Computation time on a serial computer}}{\text{Computation time on the parallel computer}}$$

To be fair, we compare the best serial algorithm for a specific problem with the best parallel algorithm for the same problem, whether or not the parallel algorithm is an adaptation of the serial algorithm. For a parallel architecture with a processor or functional multiplicity of  $N$ , the ideal speed-up ratio is  $N$ ; this is rarely obtainable. When the speed-up ratio is  $kN$ ,  $k < 1$ , but not much less, we have a problem very well-suited to parallel computation. Speed-up ratios of  $kN/\log_2 N$  are less desirable, while speed-up ratios of  $k \log_2 N$  are simply inadequate. In the present thesis, we create the best parallel algorithms by adapting the best sequential algorithms. As a rough approximation, we consider the cycle time of the parallel computers to be equal to that of the sequential machines. Hence, we measure our speed-up ratios as number of coefficient multiplications (which does not change from the sequential case)

divided by the number of cycles required to perform the computation on the parallel machine. For both parallel architectures considered here, the speed-up ratios approach the theoretical ideal.

CHAPTER VIII

MULTIPROCESSOR OPTION  
(MULTIPROCESSOR E)

## CHAPTER VIII

MULTIPROCESSOR OPTION  
(MULTIPROCESSOR E)

In this section we describe the parallel architecture envisaged, give a formal specification of the corresponding parallel algorithm, and obtain a lower bound on the speed-up ratio for this system. We choose to make a parallel adaptation of the sequential algorithm BINE, and call the resulting parallel algorithm: 'Multiprocessor E'. The basic ideas in adapting BINE are not radically different from those involved in adapting BINF; the description and analysis of Multiprocessor E, however, is somewhat simpler. In any case, ease of parallel adaptation is more important than the relatively minor time difference between BINE and BINF.

8.1 DESCRIPTION OF THE ARCHITECTURE

The parallel machine consists of a control unit with the processing capabilities of a conventional computer,  $N$  processing units or slave processors capable of decoding a limited set of special-purpose instructions set into memory by the control unit, and a large random-access central memory to which both the control processor and all slave processors have full access. Scheduling and memory management are the job of the control processor, who allocates himself a reserved section of central memory. The slave processors read their instructions from other reserved sections of central memory, properly

initialized by the control processor. Let us take  $N$ , the number of slave processors, to be  $N = 2^5$ .

The block diagram of the system is:

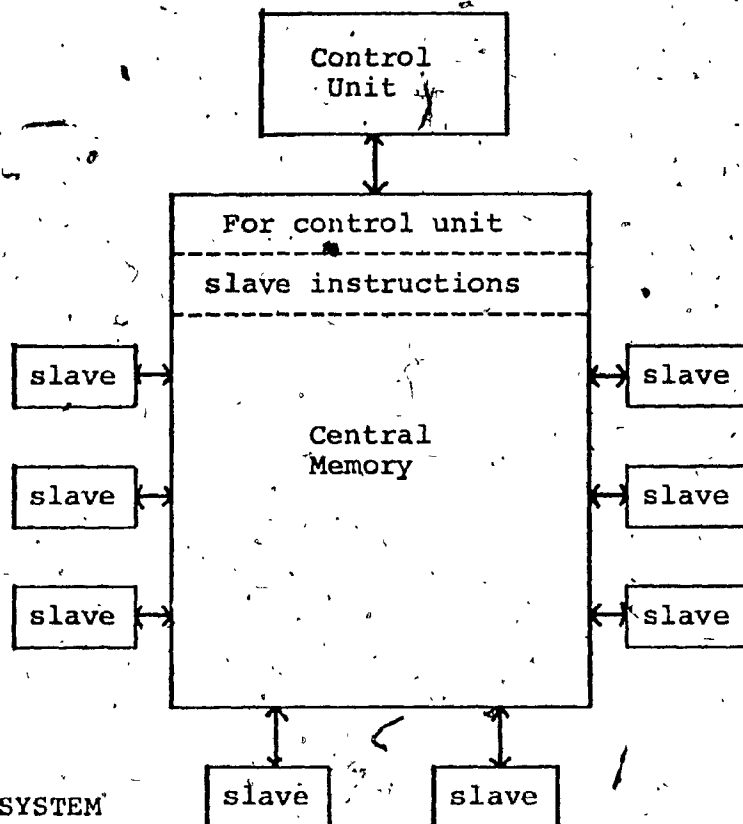


FIG. 8.1 SYSTEM DIAGRAM.

The non-reserved portion of central memory is used for the storage of intermediate results generated by the computation, that is, by any of the slave processing units. Use of central memory is essentially additive, with more and more results being added to an initially empty store. This exploits the feature of dynamic programming in BINE in which solutions to smaller subproblems are parts of solutions to larger subproblems, as discussed above. One overwriting technique,

however, is employed: in the computation of  $g_1^r \cdot \binom{n}{r} g_2^{n-r}$ , the space ultimately allocated to hold the final result is initially allocated to hold  $\binom{n}{r} g_2^{n-r}$ . This allows a clean separation within the parallel algorithm of multiplication by binomial coefficients, and multiplication by the (remaining) polynomial.

We can give a rough idea of the kind of special-purpose instruction decoded by the slave processing units. Given the character of algorithm BINE, how might  $N$  independent processors cooperate to perform the overall computation? The latter consists of a number of list constant products, each of which, sequentially, is the multiplication of a list (polynomial) by a constant (a constant product) followed by the multiplication of the resulting list by another list (a list product.) A set of list constant products therefore, may be broken into a set of constant products, and a set of list products; we require that a processor may be set to do an arbitrary amount of the work required to compute a set of constant products, or a set of list products. These lists are, of course, real lists in the linked-list sense. A sublist of a list is itself a list. A list may be specified by giving both its lead element and its length. The special purpose instructions, or rather sets of special-purpose instructions, for the slave processing units should now be clear. One slave unit, as part of computing a set of constant products, might be instructed to multiply each of several lists by one of several constants. Another slave unit, as part of computing a set of list products, might be instructed to multiply each of several lists by

one of several other lists. Operands for these operations are specified more or less as described above; the target areas in central memory to which the final results are to be written must also be carefully specified. This specification is an overhead function assumed by the control unit.

The total computation of BINE consists of a number of sets of list constant products, one such set for each node of the term-group tree. Each set of list constant products consists of a set of constant products, and a set of list products. Each set of constant products, and each set of list products consists of a number of elementary (because monomial) and hence independent (multiplication) operations. The assumption that a slave processor may be set to do an arbitrary number of elementary operations implies that  $N$  independent slave processors may divide the total computation among themselves and yield a speed-up not radically different from  $N$ . We define the idea of an  $N$ -split of  $m$  subtasks  $S_i$ , where the subtask  $S_i$  consists of  $n_i$  elementary operations. Let

$$W = \sum n_i \quad \text{and} \quad Q = \lceil W/N \rceil$$

Create  $N$  (new) subtasks  $T_i$ , of maximum size  $Q$ , in the following way. Lay out the  $m$  (original) subtasks  $S_i$  in linear order. Count off  $Q$  elementary operations from the start of  $S_1$ ; this is the (artificial) subtask  $T_1$ . Continue in the same fashion to obtain the (artificial) subtasks  $T_2$  through  $T_N$ . Assign each of the  $N$  subtasks  $T_i$  to one of the  $N$



slave processors. This completes the N-split. In the present context, the  $m$  subtasks  $S_i$  are either  $m$  constant products, or  $m$  list products. That is, we make an N-split of either a set of constant products or a set of list products, and this some number of times until the total computation is completed. We now make this more formal.

The precise operation of a multiprocessor N-split may be explained as follows. For each value of  $n$ , there is an abstract object called the power group triangle, which is used in BINE to compute all powers from 1 to  $n$  of  $a + b$  given all powers from 1 to  $n$  of each of  $a$  and  $b$ . For example, when  $n = 4$ , the triangle is:

$$\begin{array}{l}
 a^4 + b^4 + a^3 \cdot 4b + b^3 \cdot 4a + a^2 \cdot 6b^2 \\
 a^3 + b^3 + a^2 \cdot 3b + a \cdot 3a^2 \\
 a^2 + b^2 + a \cdot 2b \\
 a + b
 \end{array}$$

FIG. 8.2 POWER GROUP TRIANGLE (pgt) FOR  $n = 4$

In a pgt, there are  $\binom{n}{2}$  constant products, and  $\binom{n}{2}$  list products. The number of elementary operations in each product is a computable function of  $s = \text{size}(a) = \text{size}(b)$ . That is, we have two sets of  $\binom{n}{2}$  subtasks  $S_i$ , where we can evaluate the  $n_i$  for each  $S_i$  by knowing  $s$ . That is, for each node in

the term-group tree, we are able to perform an N-split (slave processor allocation) for all the binomial work in the pgt, and then a second N-split for the remaining nonbinomial work. The  $n_i$  are polynomial functions of  $s$  which are fixed once and for all for a given value of  $n$  (the  $n$  in  $f^n$ ), but which must be re-evaluated by the control unit each time the value of  $s$  (equivalently, tree level) changes. Once the values of the  $n_i$  are known, the N-split (slave processor allocation) proceeds as described above.

## 8.2 DESCRIPTION OF THE ALGORITHM

### Assumptions:

Let  $t$ , i.e.,  $\text{size}(f)$ , and  $N$ , i.e., the slave processor multiplicity, both be powers of two, with  $N \geq t$ . The changes to the algorithm when  $t > N$  are trivial.

### Step 1 - Creation of the Term-Group Tree

The control unit, in its section of central memory, creates a binary term-group tree for the original polynomial  $f$ . All of  $f$  goes into the root, the two halves of  $f$  go into the two subnodes of the root, and so on, recursively, with the larger half always in the left subnode. A directory is maintained which, as the computation proceeds, for each node other than the root, and each power from 1 to  $n$ , points to the list which contains the specified power of the specified subpolynomial.

## Step 2 - Processing of the Terminal Nodes

The control unit assigns one slave processor to each of the  $t$  terminal nodes, and lets the other  $N-t$  slave processors sit idle. All powers from 2 to  $n$  of each term of the original polynomial are computed. This completes the processing of the terminal nodes. The list pointers are automatically retained in the control-unit directory.

## Step 3 - Processing of the Interior Nodes

For each level of the term-group tree from  $k-1$  to 1, in that order, the control unit causes all powers from 2 to  $n$  of all nodes on that level to be computed, taking advantage of the powers already available on the next lower level. The fundamental strategy for computing a group of powers of a node, given the groups of powers of the subnodes, is expressed in the structure of the power group triangle, discussed previously. At level  $j$  there are  $2^j$  nodes, with  $j < k$ . The control unit assigns  $M = N/2^j$  slave processors to each of the  $2^j$  nodes at level  $j$ .  $M$  slave processors cooperate to process one product group triangle, i.e., one node. The control unit evaluates the time complexity of the constant products and list products involved in processing nodes at this level. The control unit then causes each group of  $M$  slave processors associated with a node to perform, first an  $M$ -split of all binomial work in the pgt, and then an  $M$ -split of the (remaining) non-binomial work. That is, the  $M$  processors split the

constant products, and then the list products. The  $2^j$  groups of slave processors work in parallel. At the end of this step, all powers from 2 to  $n$  of all interior nodes will have been computed. As before, the list pointers are automatically retained in the control-unit directory.

#### Step 4 - Processing of the Root Node

At the root level, all  $N$  slave processors are assigned to compute  $f^n$ , given the availability of the powers from 1 to  $n$  of the two sub-polynomials  $f_1$  and  $f_2$ . Although only the  $n^{\text{th}}$  power of the root is computed, the same strategy may be applied. (We merely consider the binomial-expansion of the root, written in accord with the smaller idea, to be a degenerate pgt.) As before, then, the control unit causes the group of  $N$  slave processors to perform, first an  $N$ -split of the  $n-1$  constant products of the root pgt, and then an  $N$ -split of the  $n-1$  list products. This completes the processing of the root.

Multiprocessor E, like its sequential counterpart, BINE, makes full use of the design decisions: even splitting, multi-level splitting, binary merge (dynamic programming), and smaller. These are the decisions which fix the structure of the term-group tree, and of the product group triangle. The latter merely expresses how groups of powers of polynomials are to be computed from groups of powers of subpolynomials via binomial-expansion using the smaller idea; it is a kind of

abbreviated algorithm description. The term-group tree, a multilevel, even-splitting expansion of the original polynomial, serves as a directory into the various lists and sublists generated during the computation. The pgt's are processed by the slave processors under the control of the control unit, which also manages the term-group tree.

### 8.3 ANALYSIS OF THE SPEED-UP RATIO

If a subcomputation containing  $s$  independently-allocatable units of work be divided ( $p$ -split) among  $p$  independent processors, then the speed-up is given by  $s/[s/p]$ , where  $s$  measures the time for the serial computation, and  $[s/p]$  measures the longest time taken by any of the  $p$  cooperating processors. The speed-up is less than or equal to  $p$ . If two subcomputations, with  $s_1$  and  $s_2$  units of work, respectively, are successively  $p$ -split, then the speed-up is given by

$$\frac{s}{([s_1/p] + [s_2/p])}$$

where

$$s = s_1 + s_2$$

In general,

$$[s_1/p] + [s_2/p] \geq [s/p]$$

That is, in order to maximize the speed-up of the entire computation, one should minimize the number of sub-computations which

are p-split. Multiprocessor E uses variable p-splitting insofar as M(the processor multiplicity) is a function of tree level, ranging from 1 at level k to N at level 0. As the monomial products of the computation are independently allocatable, the speed-up ratio of Multiprocessor E is given by  $E(t,n)$  divided by the sum of the longest time taken each time a p-split occurs. The number and multiplicity of the p-splits are entirely fixed by the algorithm specification; they need only be summed, or at least approximated. This, we now do.

More precisely, we calculate  $T$ , the time taken by algorithm Multiprocessor E, where  $T$  is measured in units of the time taken by a slave processor to compute one monomial product. For each non-terminal node, there are 2 p-splits: one for the constant products, and one for the list products. To calculate the time taken to process a node, we need to know the processor multiplicity, the number of monomial products in all constant products, and the number of monomial products in all list products. For example, the time taken to process the root is

$$\begin{aligned} & \left[ \frac{2 \cdot \text{group}(t/2, r-1) + N_n \cdot \text{size}(t/2, r)}{N} \right] + \\ & + \left[ \frac{\text{size}(t, n) - 2 \cdot \text{size}(t/2, n)}{N} \right] \end{aligned} \quad (8.3.1)$$

where  $r = \lceil n/2 \rceil$ ,  $N_n = 1$  (n even) or 0 (n odd), and  $N$  is the number of slave processors available on the multiprocessor system. The time taken to process a strictly interior node is

$$\left\lceil \frac{BC(s,n)}{M} \right\rceil + \left\lceil \frac{\text{group}(2s,n) - 2.\text{group}(s,n)}{M} \right\rceil \quad (8.3.2)$$

where  $s$  is the subnode size, and  $M$  the processor multiplicity (per node) available at that level. If we add in the  $n-1$  time units needed to compute all powers from 2 to  $n$  of the terminal nodes (one processor per node), and take the sum over all levels containing strictly interior nodes, we obtain finally the total time for Multiprocessor E. It is

$$\begin{aligned} T(t,n) = & \left\lceil \frac{\text{size}(t,n) - 2.\text{size}(t/2,n)}{N} \right\rceil + \\ & + \left\lceil \frac{2.\text{group}(t/2,r-1) + N_n.\text{size}(t/2,r)}{N} \right\rceil + \\ & + \sum_{j=1}^{k-1} \left\lceil \frac{\text{group}(t/2^j,n) - 2.\text{group}(t/2^{j+1},n)}{N/2^j} \right\rceil + \\ & + \left\lceil \frac{BC(t/2^{j+1},n)}{N/2^j} \right\rceil \Bigg\} + n - 1 \end{aligned} \quad (8.3.3)$$

where, again  $r = \lceil n/2 \rceil$ ,  $N_n = 1$  ( $n$  even) or  $0$  ( $n$  odd), and  $N$  is the total number of slave processors in the system. The sum runs over levels rather than nodes because the groups of  $M$  processors (per node) run in parallel. The actual speed-up ratio is given by  $E(t,n)/T(t,n)$ . We use a simple trick to obtain a lower bound on this ratio by obtaining an upper bound on  $T(t,n)$ . In general,  $\lceil s/p \rceil < s/p + 1$ . The quantity in curly brackets is

$$\left\lceil \frac{Q_1^{(j)}}{N/2^j} \right\rceil + \left\lceil \frac{Q_2^{(j)}}{N/2^j} \right\rceil < \frac{2^j (Q_1^{(j)} + Q_2^{(j)})}{N} + 2 \quad (8.3.4)$$

But  $2^j (Q_1^{(j)} + Q_2^{(j)})$  is the number of coefficient multiplications which occur at level  $j$ . (The quantity in parentheses is the number per node.) The same reasoning applies at the root level, that is,

$$\left\lceil \frac{R_1}{N} \right\rceil + \left\lceil \frac{R_2}{N} \right\rceil < \frac{R_1 + R_2}{N} + 2 \quad (8.3.5)$$

Here,  $R_1 + R_2$  is the number of coefficient multiplications at the root level. The time taken to process the terminal nodes, namely,  $n-1$ , may be written as

$$n - 1 = \frac{t(n-1)}{N} + \frac{(N-t)(n-1)}{N} \quad (8.3.6)$$

Substituting the previous (in)equalities in the formula for  $T(t,n)$  gives

$$T(t,n) < \frac{E(t,n)}{N} + 2k + \frac{(N-t)(n-1)}{N} \quad (8.3.7)$$

or

$$\frac{E(t,n)}{T(t,n)} > N \cdot \frac{E(t,n)}{E(t,n) + 2kN + (N-t)(n-1)} \quad (8.3.8)$$

Take  $N = 2t$  for definiteness. For large  $t$  or large  $n$ ,  $t(4k+n-1)$  is negligible in comparison with  $E(t,n)$ . That is, for large  $t$  or large  $n$ ,  $E(t,n)/T(t,n)$ , the speed-up ratio,



for algorithm Multiprocessor E, approaches the theoretical ideal of  $N$ , the number of processors. The only counterbalancing factor is the necessary overhead (assumed by the control unit) in properly instructing the slave processors. The most difficult and expensive operation here is probably determining the starting addresses of the various sublists assigned to the slave processors for processing.

As it now stands, algorithm Multiprocessor E does not have a space complexity as low as the more sophisticated implementations of the sequential algorithm BINE. An algorithm which consistently computes the constant products in a pgt before the list products must have space to store those constant products. This consistent approach, of course, allows a consistent and simplified p-splitting strategy. At the lower levels of the tree, one may write the constant products into the space which will subsequently contain the list products. At the lower levels, then, the space complexity does not change. At the higher levels, though, where the answers are written to disc directly, more storage will be required for the intermediate results. There is, however, no reason why a sophisticated version of Multiprocessor F could not approach the space complexity of the sequential algorithm BINF. One would simply require a more elaborate p-splitting strategy. Thus, we do not regard Multiprocessor E as in any way an ideal or optimal multiprocessor algorithm difficult to improve upon; it is merely a very clear indication how a

multiprocessor parallel architecture could be exploited to yield a dramatic speed-up of the sequential computation. There is no doubt, however, that Multiprocessor E could be much improved with respect to space complexity.

CHAPTER IX

ASSOCIATIVE-PROCESSOR OPTION  
(ASSOCIATIVE F)

CHAPTER IX  
ASSOCIATIVE-PROCESSOR OPTION  
(ASSOCIATIVE F)

Once more, in this section we describe the parallel architecture envisaged, give a formal specification of the corresponding parallel algorithm, and obtain a lower bound on the speed-up ratio for the system. This time, however, we choose to make a parallel adaptation of the sequential algorithm BINP, and call the resulting parallel algorithm: Associative F.

9.1 DESCRIPTION OF THE ARCHITECTURE

The parallel machine consists of (1) a control unit with the processing capabilities of a conventional computer, (2) a large associative or content-addressable memory in which each memory cell contains both a tag field and a term (monomial) field, and retrieval is by tag value, (3) a moderately large term buffer for storing intermediate results, and finally, (4) a parallel processing array, that is, an array of processing elements, in which each processing element, properly initialized, computes one monomial product in parallel with all other processing elements. Each element of the parallel processing array (PPA) contains a tag field and two term fields,  $t_1$  and  $t_2$ . A term field may contain one term (monomial). When the PPA is fired, each element which has been loaded computes the product  $t_2 := t_1 * t_2$ ; the resulting

values of the  $t_2$  fields may be routed either to the term buffer or to the associative memory.

The block diagram for the simplest form of this system is as follows:

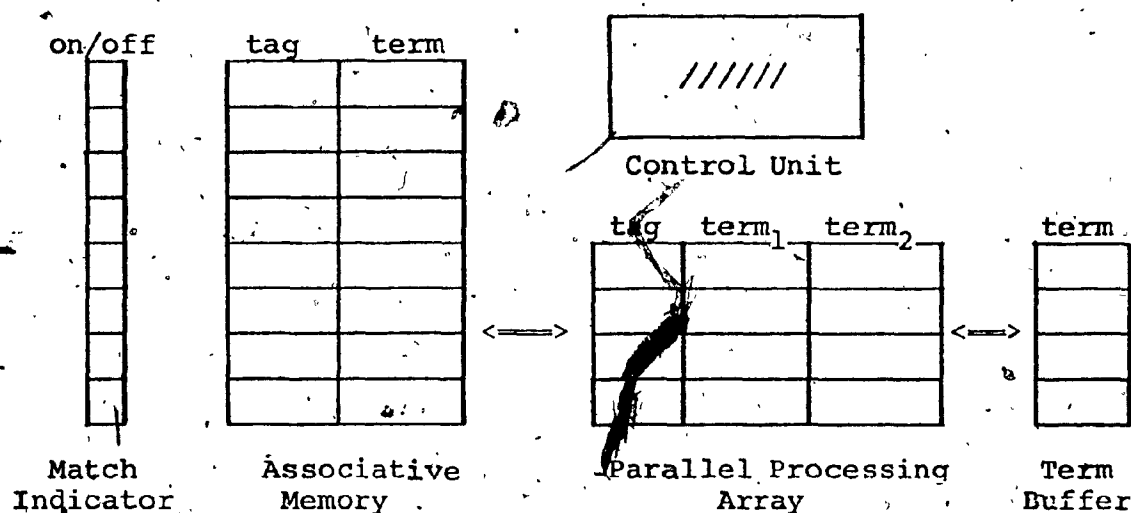


FIG. 9.1 SYSTEM DIAGRAM

The control unit is in communication with all other system units. When the control unit sets a tag value into the command buffer of the associative memory, the entire memory is searched in parallel and any matches of tag values are automatically recorded in the one-bit-per-cell match indicator. The control unit may cause the term fields of matched associative-memory cells to be routed to the term<sub>1</sub> fields of the parallel processing array. The control unit initializes the term<sub>2</sub> fields of the PPA either by routing terms from the term buffer or by inserting terms computed within the control unit itself. The control unit also initializes the tag fields of the PPA.

Roughly speaking, a tag value identifies a term belonging to a specific power of a specific sub-polynomial, that is, to one of the various intermediate polynomials generated during the computation. With an associative memory, rather than storing and retrieving these polynomials as explicitly-linked lists, we do so by storing and retrieving the terms belonging to a particular polynomial on the basis of their identifying tag values. Just as we can manipulate linked lists by resetting the pointers, so we can manipulate tagged lists by resetting the tags. The control unit (CU) requests a specific polynomial from the associative memory (AM) by giving it the appropriate tag value; the retrieved terms are then eventually routed to the  $t_1$  fields of the PPA, where they are used in the computation. The PPA generates new terms of new polynomials in the  $t_2$  fields. If the tag fields of the PPA have been properly initialized (by the CU) with the new tag values corresponding to the new polynomials, then, after the PPA has fired, the  $t_2$  fields of the PPA, together with the corresponding tag fields, may be routed back to the AM to store the new polynomials in the AM so that they can subsequently be retrieved for further computation. It is, of course, also the responsibility of the CU to initialize the  $t_2$  fields of the PPA prior to each firing. This is one of the two ways the PPA is used in Associative F.

Clearly, there is a certain overhead in storing intermediate results in the AM and then retrieving them later for further computation. Therefore, in certain circumstances, we

choose to route the new terms computed in the  $t_2$  fields of the PPA to the term buffer, without any corresponding tag values, and to retrieve them later by the simplest form of random access. PPA results go to the AM when terminal nodes are processed, or when list products are computed. The alternative routing is used during the computation of all the constant products in any one product column of a pgt. Consider, once more, a typical pgt, this one for  $n = 5$ .

$$\begin{array}{l}
 a^5 + b^5 + a^4.5b + b^4.5a + a^3.10b^2 + b^3.10a^2 \\
 a^4 + b^4 + a^3.4b + b^3.4a + a^2.6b^2 \\
 a^3 + b^3 + a^2.3b + b^2.3a \\
 a^2 + b^2 + a.2b \\
 a + b
 \end{array}$$

FIG. 9.2 Pgt FOR  $n = 5$

Our task is to up-date the AM, substituting the power group of the father node  $a + b$  for the power groups of the two sub-nodes  $a$  and  $b$ . The first two columns of the pgt are re-tagging columns, as this is all that is required here. The remaining columns are product columns, requiring the computation of new terms by multiplications performed within the PPA. For each product column, first the constant products are sent to the term buffer, and then the list products are sent to the

associative memory.

The simplest form of Multiprocessor E will run only if the corresponding parallel machine has a generous supply of central memory. Similarly, the simplest form of Associative F requires a large associative memory, and a relatively large term buffer. Both algorithms may be refined (and complicated) when the large space requirements of the simple forms become critical. In the simplest form of Associative F, the product columns of each pgt are processed separately; all constant products for a column are accumulated in the term buffer. Therefore, to process a node, we need buffer space to store the largest collection of constant products. Let  $s$  be the subnode size. When  $s$  and  $n$  are sufficiently large, we require a buffer able to hold  $\text{size}(s,p)$  terms,  $p = \lfloor n/2 \rfloor$ . For root processing,  $s = t/2$ . The associative memory must be able to hold the intermediate results from level 1 of the term-group tree; this amounts to  $2 \cdot \text{group}(t/2, n)$  associative-memory cells, and the ability to do parallel searches on associative memories of this size. We now present a version of Associative F which is suitable for this very large and powerful parallel machine. Again, we are more concerned with showing how to exploit an associative parallel architecture than with displaying the optimal associative algorithm; there is no doubt that improvements and refinements are possible.

We suppose that the parallel processing array has a multiplicity of  $N$ , and clarify the precise operation of a



PPA N-split. Consider one product column, and the constant products within it. One polynomial is to be multiplied by one or more binomial coefficients. The set of constant products is a set of subtasks in which  $N$  elementary operations at a time can be executed in parallel. As many copies of the polynomial as there are constant products are concatenated to form one long polynomial. The first  $N$  terms of this long polynomial are loaded into the  $N$   $t_1$ -fields of the PPA. The appropriate binomial coefficients are then loaded into the  $N$   $t_2$ -fields. The PPA is fired and the results are routed to the term buffer. This process is continued until all constant products for that product column have been computed. If  $W$  is the number of elementary operations in the set of constant products, then  $\lceil W/N \rceil$  firings of the PPA will be required to process the entire set. It is an important feature of constant-product N-splits that the PPA is fired precisely once for each time it is loaded. Once the set of constant products for a product column has been accumulated in the term buffer, the list products for that column may be computed. The list-product N-splits are slightly more complicated, basically in that the PPA is fired several times for each time that it is loaded. This will now also be explained.

The list products in a product column consist of, one or more times, a distinct polynomial times another distinct polynomial. These latter distinct polynomials are the constant products which have just been computed, and which are all of

precisely the same size. In a previous terminology, we have, one or more times, an a-list times a b-list. The a-lists are of varying sizes, and are available in the AM; the b-lists are all of the same size, and are available in the term buffer. All the a-lists of the product column are concatenated to form one long polynomial. The first  $N$  terms of this long polynomial are loaded into the  $N$   $t_1$ -fields of the PPA. The appropriate first terms of the various b-lists are then loaded into the  $N$   $t_2$ -fields. The PPA is fired and the results are routed to the AM. The process of loading the  $t_2$  fields and firing the PPA is continued until the b-lists are exhausted. (They will all be exhausted simultaneously.) The next  $N$  terms of the long polynomial are loaded into the  $t_1$  fields, and the whole process of  $t_1$ -loading,  $t_2$ -loading, and PPA-firing is continued until all list products for that product column have been computed. If  $W$  is the number of terms in the long polynomial, and  $S$  is the b-list size, then  $S \cdot \lceil W/N \rceil$  firings of the PPA will be required to process the entire set of list products for the product column. More exactly now, the PPA is fired precisely  $S$  times for each time the  $t_1$  fields of the PPA are loaded.

## 9.2 DESCRIPTION OF THE ALGORITHM

### Assumptions:

Let  $N$ , the PPA multiplicity, be greater than or equal to  $t$ , i.e., size ( $f$ ). The changes to the algorithm when  $t > N$

are trivial. Moreover, let  $B$ , the buffer size, be sufficient to hold the largest collection of constant products of any product column. When this condition is not met, systematic modifications to Associative  $F$  can be made which allow the computation to go through, but which, naturally, reduce the speed-up ratio.

#### Step 1 - Creation of the Term-Group Tree

This is essentially a book-keeping operation. Using its own private random-access memory, the control unit creates the term-group tree, not by physically storing terms of the polynomial, but rather by allocating tag values which are sufficient to uniquely identify specific powers of specific subpolynomials. As in the multiprocessor case, the term-group tree functions as a directory into the various lists (polynomials) stored in the AM. The term-group tree essentially specifies the node-subnode relationship, and gives the tag values for all nodes and all powers concerned.

#### Step 2 - Processing of the Terminal Nodes

The control unit gates the  $t$  terms of the original polynomial into  $t$  of the  $t_2$  fields of the PPA.  $N-t$  processing units sit idle throughout this step. The corresponding tag values are initialized by the CU and, immediately, terms together with tag values are gated to the AM. Next, the  $t$  terms are copied into the matching  $t_1$ -fields of the PPA. The latter is fired  $n-1$  times ( $t_2 := t_1 * t_2$ ), creating

the powers from 2 to  $n$  of the terminal nodes. The tag values are initialized prior to each firing, and terms plus tags are gated to the AM after each firing. The PPA can send to the AM only from its  $t_2$  fields, and can receive from the AM only in its  $t_1$  fields. After  $n-1$  firings of the PPA, all terminal node processing is complete.

### Step 3 - Processing of the Lower Interior Nodes

For each level of the term-group tree from  $k-1$  to 2, in that order, all powers from 2 to  $n$  of all nodes at that level are computed. As Associative F uses distribution, level 1 is treated separately. The fundamental strategy for computing a group of powers of a node, given the groups of powers of the two sub-nodes, is as follows. The entire PPA is allocated to one node at a time. The pgt for that node is processed product column by product column, let us say from left to right. Each product column consists of some number  $m$  of products of the form a-list times b-list. The a-lists are distinct polynomials; the b-lists are distinct binomial coefficients times a unique polynomial associated with the product column. All b-lists for the column are computed and stored in the term buffer using the constant-product N-split described earlier; the unique polynomial is retrieved from the AM. Next the  $m$  list products in the column are computed and stored in the AM using the list-product N-split described in the same place; the  $m$  a-lists are retrieved from the AM as needed to load the  $t_1$  fields of the PPA. Prior to each firing of the PPA,

the CU initializes the tag fields with appropriate tag values, and the  $t_2$  fields with terms taken from the term buffer. After each firing, the new values of the  $t_2$  fields, and the corresponding values of the tag fields (which identify the lists to which the new terms belong) are routed to the AM. After all product columns of all nodes at a level  $j$  have been processed, the CU causes all terms at level  $j + 1$  to be retagged, making the lists at the lower level part of the new level, and thus completing the process of writing the power groups of all nodes at level  $j$  into the AM. At the end of step 3, the AM contains exclusively the power groups of all nodes at level 2.

#### Step 4 - Processing of the Nodes at Level 1

Processing a pgt at level 1 is structurally identical to processing a pgt at a lower level. The difference is precisely that, in some places, distribution coefficients are substituted for binomial coefficients before the computation begins. If a binomial expansion at level 1 will be used in a b-list at level 0, then all binomial coefficients in that expansion, if any, must be substituted. For example, suppose that  $a, b, c$ , and  $d$  are the nodes at level 2, and that  $(a+b)^2 \cdot 6(c+d)^2$  is required in the binomial expansion of the root, namely  $[(a+b) + (c+d)]^4$ . As part of computing the b-list  $6(c+d)^2$  directly, we substitute  $c \cdot 12d$  for  $c \cdot 2d$  in the pgt for  $c + d$ , and so on. We defer all binomial coefficient multiplications associated with retagging columns, e.g.,  $6(c^2+d^2)$ , to Step 5.

In consequence, the retagging which normally follows the processing of all product columns of all nodes at a level will have to be modified somewhat so as to be able to retrieve, say,  $c^2$  and  $d^2$  separately from  $12cd$ . At the end of Step 4, we have computed the a-lists and some terms of the b-lists to be used at level 0.

#### Step 5 - Processing of the Root Node

As before, we write

$$f^n = f_1^n + f_2^n + \sum_{i=1}^{n-1} a\text{-list}_i \cdot b\text{-list}_i.$$

This is the usual binomial expansion in which the binomial coefficients have been included in the b-lists. The polynomials  $f_1^n, f_2^n$ , and all a-lists have been computed. We use a single constant-product N-split to perform all the binomial coefficient multiplications necessary to complete the computation of the b-lists. We load the first two b-lists into the term buffer. We concatenate the first two a-lists to form a long polynomial. We use a single list-product N-split to compute the first two list products; the answers may be written to disc. We continue in this way, two-by-two, until the  $n-1$  list products have been computed. This completes the processing of the root.

### 9.3 ANALYSIS OF THE SPEED-UP RATIO

We can obtain the speed-up ratio for Associative F by dividing the number of elementary operations in the sequential algorithm (BINF) by the number of firings of the PPA in the parallel algorithm. We denote these two quantities by  $F(t,n)$  and  $P(t,n)$ , respectively. We assess the speed-up ratio  $F(t,n)/P(t,n)$  by first calculating and then approximating the function  $P(t,n)$ . There are  $n-1$  firings of the PPA to process the terminal nodes. We calculate the number of firings required to process a node, at any level from  $k-1$  to 2, when the sub-node size is  $s$ . We calculate first the number of firings for all binomial work in a pgt. This is given by

$$BW(s) = \sum_{i=1}^{n-1} \left\lceil \frac{1}{N} \cdot (n-i) \cdot \binom{s+u-1}{u} \right\rceil, \text{ where } u = \lceil i/2 \rceil \quad (9.3.1)$$

This is just the sum over product columns of ceiling of  $1/N$  of the size of all b-lists in that column. Next, we calculate the number of firings for all non-binomial work. This is given by

$$NBW(s) = \sum_{i=1}^{n-1} \binom{s+u-1}{u} \cdot \left\lceil \frac{1}{N} \sum_{j=v}^{n-u} \binom{s+j-1}{j} \right\rceil, \text{ where } u = \lceil i/2 \rceil \quad (9.3.2)$$

and  $v = \lceil (i+1)/2 \rceil$ . This is just the sum over product columns of the size of the b-list times ceiling of  $1/N$  of the size of all a-lists in that column. We now have, for each node, the number of firings in all constant-product N-splits, and the number in all list-product N-splits.

The two formulas  $BW(s)$  and  $NBW(s)$  need to be summed over all interior nodes from level  $k-1$  to level 2 for the total number of firings of the PPA in the classical pgt work.

Since the processing at level 1 differs only in the substitution of distribution coefficients, the sum may be extended to level 1. The number of firings in the single constant-product  $N$ -split at the root level is given by

$$\left\lceil \frac{1}{N} \cdot [4 \cdot \text{group}(t/4, r-1) + 2N_n \cdot \text{size}(t/4, r)] \right\rceil \quad (9.3.3)$$

where  $r = \lceil n/2 \rceil$  and  $N_n = 1(0)$  when  $n$  is even (odd). This is just ceiling of  $1/N$  times all root binomial work in either of the  $F$  algorithms (BINF or Associative  $F$ ). The number of firings in the  $r-1$  or  $r$  list-product  $N$ -splits at the root level is given by

$$\sum_{j=1}^{r-1} \binom{t/2+j-1}{j} \cdot \left\lceil \frac{1}{N} \cdot 2 \binom{t/2+n-j-1}{n-j} \right\rceil + N_n \cdot \binom{t/2+r-1}{r} \cdot \left\lceil \frac{1}{N} \binom{t/2+r-1}{r} \right\rceil \quad (9.3.4)$$

where  $r$  and  $N_n$  have the same meanings as above. If we perform the indicated summation of  $BW(s)$  and  $NBW(s)$ , and add the firings for the root node and the terminal nodes, we obtain finally an expression for  $P(t, n)$ . This is given by



$$\begin{aligned}
P(t,n) = & \sum_{j=1}^{r-1} \binom{t/2+j-1}{j} \cdot \left\lceil \frac{2}{N} \binom{t/2+n-j-1}{n-j} \right\rceil + N_n \cdot \binom{t/2+r-1}{r} \\
& \cdot \left\lceil \frac{1}{N} \binom{t/2+r-1}{r} \right\rceil + \left\lceil \frac{1}{N} \cdot [4.\text{group}(t/4, r-1) + \right. \\
& \left. + 2N_n.\text{size}(t/4, r)] \right\rceil + \sum_{j=1}^{k-1} 2^j [BW(t/2^{j+1}) + \\
& + NBW(t/2^{j+1})] + n - 1 \tag{9.3.5}
\end{aligned}$$

As before, we obtain a lower bound on the speed-up ratio  $R(t,n)/P(t,n)$  by obtaining an upper bound on the quantity  $P(t,n)$ . In general,  $\lceil s/p \rceil < s/p + 1$ . We make systematic substitutions of this inequality for each occurrence of the ceiling function to obtain an upper bound of the form  $P(t,n) < F(t,n)/N + Q(t,n)$ . The contribution to  $Q(t,n)$  from the root is just  $\text{group}(t/2, r-1) + N_n.\text{group}(t/2, r) + 1$ . As explained in the multiprocessor analysis, the terminal nodes give rise to a contribution of  $(N-t)(n-1)/N$ . If we refer back to the form of  $BW(s)$  and  $NBW(s)$ , we see that each interior node makes a contribution of  $n-1 + 2.\text{group}(s, r-1) + N_n.\text{size}(s, r)$ , where  $s$ , as always, is the sub-node size. The term  $n-1$  comes from substituting the ceiling inequality in  $BW(s)$ , the remaining terms from substituting the same inequality in  $NBW(s)$ .

Therefore, we may write

$$\begin{aligned}
Q(t,n) &= \text{group}(t/2, r-1) + N_n \cdot \text{size}(t/2, r) + 1 \\
&+ \sum_{j=1}^{k-1} 2^j [n-1 + 2 \cdot \text{group}(t/2^{j+1}, r-1) + \\
&+ N_n \cdot \text{size}(t/2^{j+1}, r)] + (N-t)(n-1)/N \quad (9.3.6)
\end{aligned}$$

The summation may be written as

$$\begin{aligned}
(n-1)(t-2) + \frac{t}{2} \cdot \sum_{m=0}^{k-2} 2^{-m} [2 \cdot \text{group}(2^m, r-1) + \\
+ N_n \cdot \text{size}(2^m, r)] \quad (9.3.7)
\end{aligned}$$

Using the closed forms of  $\text{group}(s,n)$  and  $\text{size}(s,n)$  this is

$$(n-1)(t-2) + \frac{t}{2} \cdot \sum_{m=0}^{k-2} \sum_{j=1}^r b_j^{(r)} (2^m)^{j-1} \quad (9.3.8)$$

where

$$b_j^{(r)} = \frac{2}{(r-1)!} \left[ \begin{matrix} r \\ j+1 \end{matrix} \right] + \frac{N_n}{r!} \left[ \begin{matrix} r \\ j \end{matrix} \right], \quad \text{and} \quad \left[ \begin{matrix} r \\ r+1 \end{matrix} \right] = 0$$

Simplifying gives

$$\begin{aligned}
(n-1)(t-2) + \frac{t}{2} \cdot b_1^{(r)} \cdot k + \frac{t}{2} \cdot \sum_{j=1}^{r-1} b_{j+1}^{(r)} \cdot \frac{(t/2)^{j-1}}{2^{j-1}} \quad (9.3.9)
\end{aligned}$$

Finally then

$$\begin{aligned}
Q(t,n) = & \text{group}(t/2, r-1) + N_n \cdot \text{size}(t/2, r) + \\
& + (N-t)(n-1)/N + \frac{t}{2} \cdot \sum_{j=1}^{r-1} b_{j+1}^{(r)} \cdot \frac{(t/2)^{j-1}}{2^{j-1}} + \\
& + (n-1)(t-2) + 1 + \frac{t}{2} \cdot b_1^{(r)} \cdot k
\end{aligned} \tag{9.3.10}$$

Since  $P(t,n) < F(t,n)/N + Q(t,n)$ , we have the following lower bound on the speed-up ratio for Associative F.

$$\frac{F(t,n)}{P(t,n)} > N \cdot \frac{F(t,n)}{F(t,n) + N \cdot Q(t,n)} \tag{9.3.11}$$

This speed-up ratio is, clearly, less attractive than the speed-up ratio for Multiprocessor E. The leading term of  $Q(t,n)$  is given by

$$\frac{1}{P!} \left(\frac{t}{2}\right)^P \cdot \left[1 + \frac{2-N_n}{2^{P-1}-1}\right], \text{ where } P = \lfloor n/2 \rfloor \tag{9.3.12}$$

In comparison, the leading term of  $F(t,n)$  is  $t^n/n!$ . Therefore, for large  $t$  and large  $n$ ,  $Q(t,n)$  will be negligible in comparison with  $F(t,n)$ . It follows that, asymptotically, the speed-up ratio for Associative F approaches the theoretical ideal. The quantity  $N \cdot Q(t,n)$  is vastly larger than  $2kn + (N-t)(n-1)$ , the corresponding quantity in the formula for the lower bound on the speed-up ratio for Multiprocessor E. It cannot automatically be concluded that the multiprocessor parallel architecture is superior to the associative-processor architecture for this computation; one would need to make a careful study

of the hidden overhead in Multiprocessor E. It seems that the overhead in Associative F is considerably less. This may ultimately tip the scales in favour of Associative F.

Several modifications and improvements of algorithm Associative F suggest themselves. One could adopt a unified strategy by processing pairs of product columns in the pgt; this is the way list products are computed at the root level in the present version. The pairwise strategy would reduce the contribution of each node to  $Q(t,n)$  to  $n-1 + \text{group}(s,r-1) + N_n \cdot \text{size}(s,r)$ , which affects the coefficient of the leading term of  $Q(t,n)$ . (In (9.3.12),  $2-N_n$  becomes 1). In case the term buffer is not large enough, a simple modification of the architecture, namely, providing separate match indicators for a-lists and b-lists, allows one to segment the subcomputations as they are currently defined, and have piecemeal loading of parts of a-lists and parts of b-lists, alternately, without additional associative searching. Finally, in choosing between Multiprocessor E and Associative F, one must consider the machine cost. The processing elements of the PPA are simpler than the slave processors of the multiprocessor machine. Thus, it is entirely possible that, from an economic standpoint, one could achieve much larger multiplicity with the associative architecture. We believe that Associative F is a strong argument in favour of the very great suitability of an associative parallel architecture for this class of computations, more suitable, in fact, than the multiprocessor architecture.

We have been careful not to claim that algorithm Associative F is the optimal way to exploit the proposed special-purpose associative-processor architecture. Yet clearly it is a good way, for the following reason. We make best use of the PPA by minimizing the number of times it is fired when it is less than completely full. Associative F adopts the strategy of processing the list products in a pgt at most two product columns at a time. This ensures that the b-lists in the set of list products are all of precisely one size, say,  $b$ . It will require some number of loads of the  $t_1$  fields of the PPA to exhaust the a-lists in the set of list products. For each such load, except possibly the last, there will be  $b$  firings of the PPA which use its capacity to the fullest. Moreover, the initializations of the  $t_2$  fields of the PPA prior to each firing are straightforward: Let  $N_i$   $t_1$ -fields of the PPA contain all or part of an a-list,  $a_i$ ; the  $N_i$  matching  $t_2$ -fields are all filled with the next term of the corresponding b-list,  $b_i$ . With trivial random-access initialization of the  $t_2$  fields, we get the benefit of  $b$  firings of all  $N$  cells of the PPA before the next load from the AM. We have reason to believe, then, that we are making good use of this parallel machine, and that processing two product columns at a time is the best we can do.

The space requirements for Associative F are not absolute, in the following sense. Normally, to process a set of list products by the method outlined above, one needs a term buffer able to hold all the b-lists belonging to the set,

and all at once. When a buffer of this size is not available, the computation may be performed in a piecemeal fashion, with a resultant decrease in speed-up ratio. The discussions relative to the space complexity of the sequential algorithm BINF show that one could not hope to implement the parallel algorithm Associative F, as it now stands, with less than  $\text{size}(t/2, p) + 4.\text{group}(t/4, n-1)$  cells of associative memory, nor less than  $\text{size}(t/2, p)$  cells of term buffer, where  $p = \lfloor n/2 \rfloor$ . The two requirements are additive. Algorithm modification amounting to space-time trade-off has already been mentioned; given the lower cost of buffer cells, it is probably not a good idea to make up for insufficient buffer size by increasing the size of the associative memory. A somewhat generous estimate for the space complexity of Associative F, then, if we combine the two forms of memory, is given by

$$2.\text{size}(t/2, p) + 4.\text{group}(t/4, n-1), p = \lfloor n/2 \rfloor$$

(9.3.13)

CHAPTER X  
CONCLUSION

## CHAPTER X

## CONCLUSION

We have restricted ourselves to the problem of the symbolic computation of integer powers of completely or almost completely sparse multivariate polynomials. Six new algorithms for this problem, namely, the four sequential algorithms BINC, BIND, BINE, and BINF, and the two parallel algorithms Multiprocessor E and Associative F, have been proposed and investigated. The two parallel algorithms are specifically intended to be run on two special-purpose parallel machines, also discussed in this thesis. All six algorithms are based on the idea of using binomial expansion as the fundamental and exclusive tool for computing powers of polynomials, for the desired power of the original polynomial, for powers of subpolynomials (other than monomials) arising in the original binomial expansion, and so on, recursively. Previous analysis did suggest the superiority of binomial expansion as a general approach. However, the previous best sequential binomial-expansion algorithm, namely, BINB, did not carry the binomial-expansion approach through systematically, i.e., recursively, and so did not realize the full benefits of the binomial-expansion approach. The four new sequential algorithms are successive refinements and improvements of the systematic binomial-expansion approach.

The main conclusions for the sequential algorithms may be summarized as follows. Both the time complexity and the



space complexity of the algorithms depend on the design decisions relating to polynomial splitting, subpolynomial powering, and cross-product formation, as discussed above. When the polynomials are sparse, even splitting is to be preferred because of the substantial reduction in the cost of powering subpolynomials; this idea is used in all of these algorithms, including BINB. Repeated multiplication is not a good way to obtain powers of subpolynomials, both because of the excessive number of coefficient multiplications required, and because of the need to sort the intermediate results; this less than optimal approach is used in BINB, but in none of the new algorithms. BING and BIND both use recursion to generate powers of subpolynomials. BINE uses a modified form of recursion akin to dynamic programming. BINF uses a modified form of dynamic programming which avoids some intermediate steps. In this series, the time complexity decreases each time. We think it is very unlikely that there is something better than dynamic programming for computing the powers of subpolynomials, given the computational and cost models which have been adopted in the thesis, and discussed above.

BINC, alone among the new algorithms in this respect, does not use the rather obvious improvement of always multiplying the binomial coefficient by the smaller polynomial first; BINB also fails to take advantage of this improvement. When we analyse the time complexities of the five sequential binomial-expansion algorithms BINB, BINC, BIND, BINE, and BINF, we

find that the respective cost functions form a strictly monotonically decreasing (finite) sequence. The new algorithms have the further advantage that, when the polynomials are sparse, they do not, unlike BINB, require exponent comparisons. If one is generous in assigning a low space complexity to BINB, then only sophisticated implementations of BINE, and the standard implementation of BINF, have lower space complexities. BINF does very well from a space-complexity standpoint; the modified form of dynamic programming it uses was specifically introduced to save space, not time. We conjecture that BINF is optimal for both time and space among sequential binomial-expansion algorithms. The cost comparisons have been made using both comparison of leading terms of analytically-obtained cost functions, and straight tabulation; the two methods do not give different results.

. In the parallel case, the aim is to devise parallel algorithms in conjunction with special-purpose parallel architectures; parallel solutions to the problem of powering sparse polynomials appear not to have been studied elsewhere. The two parallel algorithms Multiprocessor E and Associative F both have speed-up ratios which, asymptotically, approach the theoretical upper limit. Although the calculated speed-up ratio of the former seems to suggest that Multiprocessor E is the superior of the two algorithms, we believe that the time complexity hidden in the Multiprocessor E overhead, and hardware costs, both shift the balance in favour of Associative F. The space complexities of the two parallel algorithms, apparently,

are not enormously different from the space complexities of their sequential counterparts. The main parallel result is, not to exhibit optimal parallel algorithms, but rather to show convincingly how very well-suited these or similar parallel algorithms, running on the two proposed parallel machines, are to the general problem of powering sparse polynomials. It seems also that good sequential algorithms for this problem may be adapted, more or less directly, to yield good parallel algorithms. The whole parallel area is wide open for further research.

This thesis has been intended as a contribution to theoretical computational complexity. It is the feeling among computer scientists today that no new algorithm can be respectably presented without some analysis of its behaviour. While not rejecting the approach which is based on empirical comparisons of (hopefully reasonable) implementations of competing algorithms, our approach to algorithm analysis has been essentially to obtain the analytically-exact cost functions, and compare them. However, we have tested the performance of algorithm BINE with a full implementation in PASCAL 6000. We state again that major portions of this thesis have previously been published in [2]; a short comparison of the earlier and later write-ups occurs at the end of Chapter 1.

REFERENCES

REFERENCES<sup>1</sup>

- [1] Aho, A., Hopcroft, J., and Ullman, J., The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974, pp.311-313.
- [2] Alagar, V., and Probst, D., "Binomial-Expansion Algorithms for Computing Integer Powers of Sparse Polynomials", in International Computing Symposium 1977, E. Morlet and D. Ribbens, eds., North-Holland, Amsterdam, 1977, pp.395-402.
- [3] Dijkstra, E.W., "Structured Programming", in Software Engineering Techniques, J.N. Buxton and B. Randell, eds., NATO Science Committee, Brussels, 1970, pp.84-88.
- [4] Fateman, R.J., "On the Computation of Powers of Sparse Polynomials", Studies in Appl.Math., Vol.53, No. 2, June 1974, pp.145-155.
- [5] Fateman, R.J., "Polynomial Multiplication, Powers and Asymptotic Analysis: Some Comments", SIAM J. Comput., Vol.3, No. 3, Sept. 1974, pp.196-213.
- [6] Flynn, M.J., "Some Computer Organizations and Their Effectiveness", IEEE Trans. on Comput., Vol. C-21, No. 9, Sept. 1972, pp.948-960.
- [7] Gentleman, W.M., "Optimal Multiplication Chains for Computing a Power of a Symbolic Polynomial", Math. of Comput., Vol.26, No.120, Oct.1972, pp.935-939.
- [8] Gentleman, W.M., "On the Relevance of Various Cost Models of Complexity", in Complexity of Sequential and Parallel Numerical Algorithms, J.F. Traub, ed., Academic Press, New York, 1973, pp.103-109.
- [9] Heindel, L.B., "Computation of Powers of Multivariate Polynomials Over the Integers", J.Comp.Syst.Sci., Vol. 6, 1971, pp.1-8.

---

<sup>1</sup>References [8],[9],[11], and [15] have not been cited in the text.

- [10] Horowitz, E., and Sahni, S., "The Computation of Powers of Symbolic Polynomials", SIAM J. Comput., Vol. 4, No. 2, June 1975, pp.201-208.
- [11] Katz, J.H., "Matrix Computations on an Associative Processor", in Parallel Processor Systems, Technologies, and Applications, L.C. Hobbs, et al., eds., Spartan Books, New York, 1970, pp.131-149.
- [12] Knuth, D.E., The Art of Computer Programming, Vol.1, Fundamental Algorithms, 2nd edition, Addison-Wesley, Reading, Mass., 1975, pp.65-67.
- [13] Parnas, D.L., "On the Design and Development of Program Families", IEEE Trans. on Soft.Eng., Vol. SE-2, No.1, Jan. 1976, pp.1-9.
- [14] Stone, H.S., "Problems of Parallel Computation", in Complexity of Sequential and Parallel Numerical Algorithms, op.cit., pp.1-15.
- [15] Thurber, K.J., and Wald, L.D., "Associative and Parallel Processors", Comput.Surv., Vol.7, No.4, Dec.1975, pp.215-255.

APPENDIX I

VALUES OF  $B(t,n)$ ,  $C(t,n)$ ,  $D(t,n)$ ,  $E(t,n)$ , AND  
 $L(t,n)$  FOR SELECTED VALUES OF  $t$  AND  $n$

T	N	B	C	D	E	L
4	4	69	98	96	68	31
8	4	550	608	586	458	322
16	4	5878	5116	4944	4400	3860
17	4	7334	6252	6065	5434	4828
18	4	8980	7613	7366	6648	5967
19	4	10961	9150	8865	8060	7296
20	4	13173	11070	10584	9692	8835
21	4	15792	13058	12544	11555	10605
22	4	18687	15357	14768	13682	12628
23	4	22068	17922	17277	16494	14927
24	4	25774	21604	20098	18818	17526
25	4	30052	24696	23265	21865	20450
26	4	34708	28193	26796	25276	23725
27	4	40029	32054	30715	29075	27378
28	4	45785	36494	35054	33294	31437
29	4	52306	41206	39839	37947	35931
30	4	59323	46441	45104	43080	40890
31	4	67212	52162	50876	48720	46345
32	4	75662	58424	57192	54904	52328
4	5	110	178	174	110	52
8	5	1254	1419	1364	1036	784
16	5	21870	18988	18412	16852	15488
17	5	28687	24403	23796	21948	20332
18	5	36664	31172	30320	28184	26316
19	5	46830	39282	38247	35823	33630
20	5	58558	49639	47714	45002	42484
21	5	73178	61715	59076	56034	53109
22	5	89846	75003	72462	69090	65758
23	5	110245	91200	88247	84545	80707
24	5	133270	114527	106632	102600	98256
25	5	161013	135572	128144	123688	118730
26	5	192060	160107	152932	148052	142480
27	5	228972	188307	181503	176199	169884
28	5	269974	221274	214134	208406	201348
29	5	318160	258144	251457	245243	237307
30	5	371338	300242	293740	287040	278226
31	5	433203	347865	341642	334456	324601
32	5	501086	401652	395528	387856	376960
4	6	162	290	284	164	80
8	6	2574	2923	2820	2140	1708
16	6	71462	62538	61084	57508	54248
17	6	98238	84484	82942	78617	74596
18	6	131020	113115	110830	105756	100929
19	6	174670	149296	146296	140473	134577
20	6	227198	196715	190672	184100	177080
21	6	295333	253107	246115	238660	230209
22	6	376143	323243	314274	305936	295988
23	6	478702	409016	397801	388580	376717
24	6	598838	532548	498956	488852	474996
25	6	748521	652920	621356	610067	593750



T	N	B	C	D	E	L
26	6	921984	797703	767598	755124	736255
27	6	1134722	970146	941871	928212	906165
28	6	1378956	1176437	1147714	1132870	1107540
29	6	1674418	1417035	1390442	1374181	1344875
30	6	2010833	1699647	1674210	1656532	1623130
31	6	2412970	2029322	2005249	1986154	1947761
32	6	2867502	2412504	2388784	2368272	2324752
4	7	226	439	430	230	116
8	7	4894	5529	5344	4104	3424
16	7	211774	188160	184740	177596	170528
17	7	304397	266707	263151	254333	245140
18	7	423376	373591	367896	357404	346086
19	7	587993	515312	507323	495157	480681
20	7	795138	706573	689254	675414	657780
21	7	1073406	947303	926066	910192	888009
22	7	1417480	1257875	1228658	1210750	1184018
23	7	1868465	1652876	1613841	1593899	1560757
24	7	2417758	2223285	2097304	2075328	2035776
25	7	3122905	2820994	2702731	2677906	2629550
26	7	3970588	3563212	3450664	3422990	3365830
27	7	5039559	4476529	4371619	4341096	4272021
28	7	6309938	5598258	5493950	5460578	5379588
29	7	7887388	6953662	6858651	6821288	6724491
30	7	9743128	8591384	8500808	8460254	8347650
31	7	12016489	10557437	10472904	10428759	10295441
32	7	14666878	12906000	12821992	12774256	12620224

APPENDIX II

VALUES OF  $E(t,n) - F(t,n)$  FOR SELECTED  
VALUES OF  $t$  AND  $n$

T	N	E(T,N)-F(T,N)
16	5	33
17	5	37
18	5	41
19	5	46
20	5	51
21	5	56
22	5	61
23	5	67
24	5	73
25	5	79
26	5	85
27	5	92
28	5	99
29	5	106
30	5	113
31	5	121
32	5	129
16	6	114
17	6	118
18	6	167
19	6	157
20	6	202
21	6	207
22	6	278
23	6	263
24	6	326
25	6	332
26	6	429
27	6	408
28	6	492
29	6	499
30	6	626
31	6	598
32	6	706
16	7	194
17	7	228
18	7	262
19	7	307
20	7	352
21	7	402
22	7	452
23	7	515
24	7	578
25	7	647
26	7	716
27	7	800
28	7	884
29	7	975
30	7	1066
31	7	1174
32	7	1282

APPENDIX III

VALUES OF  $S_B(t,n)$ ,  $S_E(t,n)$ ,  $\underline{S}_E(t,n)$ , AND  
 $S_F(t,n)$  FOR SELECTED VALUES OF  $t$  AND  $n$