Efficient Evaluation of Large Polynomials

Efficient Evaluation of Large Polynomials

Charles E. Leiserson[1], Liyun Li[2], Marc Moreno Maza[2], and Yuzhen Xie[2]

[1] CSAIL, Massachussets Institute of Technology, Cambridge MA, USA
[2] Department of Computer Science, University of Western Ontario, London ON, Canada

Abstract. Minimizing the evaluation cost of a polynomial expression is a fundamental problem in computer science. We propose tools that, for a polynomial P given as the sum of its terms, compute a representation that permits a more efficient evaluation. Our algorithm runs in $d(nt)^{O(1)}$ bit operations plus $dt^{O(1)}$ operations in the base field where d, n and t are the total degree, number of variables and number of terms of P. Our experimental results show that our approach can handle much larger polynomials than other available software solutions. Moreover, our computed representation reduce the evaluation cost of P substantially.

Keywords: *Multivariate polynomial evaluation, code optimization, Cilk++.*

1 Introduction

If polynomials and matrices are the fundamental mathematical entities on which computer algebra algorithms operate, expression trees are the common data type that computer algebra systems use for all their symbolic objects. In MAPLE, by means of common subexpression elimination, an expression tree can be encoded as a directed acyclic graph (DAG) which can then be turned into a straight-line program (SLP), if required by the user. These two data-structures are well adapted when a polynomial (or a matrix depending on some variables) needs to be regarded as a function and evaluated at points which are not known in advance and whose coordinates may contain "symbolic expressions". This is a fundamental technique, for instance in the *Hensel-Newton lifting techniques* [6] which are used in many places in scientific computing.

In this work, we study and develop tools for manipulating polynomials as DAGs. The main goal is to be able to compute with polynomials that are far too large for being manipulated using standard encodings (such as lists of terms) and thus where the only hope is to represent them as DAGs. Our main tool is an algorithm that, for a polynomial P given as the sum its terms, computes a DAG representation which permits to evaluate P more efficiently in terms of work, data locality and parallelism. After introducing the related concepts in Section 2, this algorithm is presented in Section 3.

The initial motivation of this study arose from the following problem. Consider $a = a_m x^m + \cdots + a_1 x + a_0$ and $b = b_n x^n + \cdots + b_1 x + b_0$ two *generic* univariate polynomials of respective positive degrees m and n. Let $R(a, b)$ be the resultant of a and b. By generic polynomials, we mean here that $a_m, \ldots, a_1, a_0, b_n, \ldots, b_1, b_0$ are independent symbols. Suppose that $a_m, \ldots, a_1, a_0, b_n, \ldots, b_1, b_0$ are substituted to polynomials $\alpha_m, \ldots, \alpha_1, \alpha_0, \beta_n, \ldots, \beta_1, \beta_0$ in some other variables $c_1, \ldots, c_p$. Let us denote by $R(\alpha, \beta)$ the "specialized" resultant. If these α_i's and β_j's are large, then computing $R(\alpha, \beta)$ as a polynomial in $c_1, \ldots, c_p$, expressed as the sum of its terms, may become practically impossible. However, if $R(a, b)$ was originally computed as a DAG with $a_m, \ldots, a_1, a_0, b_n, \ldots, b_1, b_0$ as input and if the α_i's and β_j's are also given as DAGs with $c_1, \ldots, c_p$ as input, then one may still be able to manipulate $R(\alpha, \beta)$.

The techniques presented in this work do not make any assumptions about the input polynomials and, thus, they are not specific to resultant of generic polynomials. We simply use this example as an illustrative well-known problem in computer algebra.

Given an input polynomial expression, there are a number of approaches focusing on minimizing its size. Conventional common subexpression elimination techniques are typical methods to optimize an expression. However, as general-purpose applications, they are not suited for optimizing large polynomial expressions. In particular, they do not take full advantage of the algebraic properties of polynomials. Some researchers have developed special methods for making use of algebraic factorization in eliminating common subexpressions [1, 7] but this is still not sufficient for minimizing the size of a polynomial expression. Indeed, such a polynomial may be irreducible. One economic and popular approach to reduce the size of polynomial expressions and facilitate their evaluation is the use of Horner's rule. This high-school trick for univariate polynomials has been extended to multivariate polynomials via different schemes [8, 9, 3, 4]. However, it is difficult to compare these extensions and obtain an optimal scheme from any of them. Indeed, they all rely on selecting an appropriate ordering of the variables. Unfortunately, there are $n!$ possible orderings for n variables.

As shown in Section 4, our algorithm runs in polynomial time w.r.t. the number of variables, total degree and number of terms of the input polynomial expression. We have implemented our algorithm in the `Cilk++` concurrency platform. Our experimental results reported in Section 5 illustrate the effectiveness of our approach compared to other available software tools. For $2 \leq n, m \leq 7$, we have applied our techniques to the resultant $R(a, b)$ defined above. For $(n, m) = (7, 6)$, our optimized DAG representation can be evaluated sequentially 10 times faster than the input DAG representation. For that problem, none of code optimization software tools that we have tried produces a satisfactory result.

2 Syntactic Decomposition of a Polynomial

Let $\mathbb{K}$ be a field and let $x_1 > \cdots > x_n$ be n ordered variables, with $n \geq 1$. Define $X = \{x_1, \ldots, x_n\}$. We denote by $\mathbb{K}[X]$ the ring of polynomials with coefficients in $\mathbb{K}$ and with variables in X. For a non-zero polynomial $f \in \mathbb{K}[X]$, the set of its monomials is $\mathsf{mons}(f)$, thus f writes $f = \sum_{m \in \mathsf{mons}(f)} c_m\, m$, where, for all $m \in \mathsf{mons}(f)$, $c_m \in \mathbb{K}$ is the coefficient of f w.r.t. m. The set $\mathsf{terms}(f) = \{c_m\, m \mid m \in \mathsf{mons}(f)\}$ is the set of the terms of f. We use $\sharp\mathsf{terms}(f)$ to denote the number of terms in f.

Syntactic operations. Let $g, h \in \mathbb{K}[X]$. We say that gh is a *syntactic product*, and we write $g \odot h$, whenever $\sharp\mathsf{terms}(g\, h) = \sharp\mathsf{terms}(g) \cdot \sharp\mathsf{terms}(h)$ holds, that is, if no grouping of terms occurs when multiplying g and h. Similarly, we say that $g + h$ (resp. $g - h$) is a *syntactic sum* (resp. *syntactic difference*), written $g \oplus h$ (resp. $g \ominus h$), if we have $\sharp\mathsf{terms}(g+h) = \sharp\mathsf{terms}(g) + \sharp\mathsf{terms}(h)$ (resp. $\sharp\mathsf{terms}(g-h) = \sharp\mathsf{terms}(g) + \sharp\mathsf{terms}(h)$).

Syntactic factorization. For non-constant $f, g, h \in \mathbb{K}[X]$, we say that $g\,h$ is a *syntactic factorization* of f if $f = g \odot h$ holds. A syntactic factorization is said *trivial* if each factor is a single term. For a set of monomials $\mathcal{M} \subset \mathbb{K}[X]$ we say that $g\,h$ is a syntactic factorization of f *with respect to* $\mathcal{M}$ if $f = g \odot h$ and $\mathsf{mons}(g) \subseteq \mathcal{M}$ both hold.

Evaluation cost. Assume that $f \in \mathbb{K}[X]$ is non-constant. We call *evaluation cost* of f, denoted by $\mathsf{cost}(f)$, the minimum number of arithmetic operations necessary to evaluate f when $x_1, \ldots, x_n$ are replaced by actual values from $\mathbb{K}$ (or an extension field of $\mathbb{K}$). For a constant f we define $\mathsf{cost}(f) = 0$. Proposition 1 gives an obvious upper bound for $\mathsf{cost}(f)$. The proof, which is routine, is not reported here.

Proposition 1 *Let $f, g, h \in \mathbb{K}[X]$ be non-constant polynomials with total degrees d_f, d_g, d_h and numbers of terms t_f, t_g, t_h. Then, we have $\mathsf{cost}(f) \leq t_f(d_f + 1) - 1$. Moreover, if $g \odot h$ is a nontrivial syntactic factorization of f, then we have:*

$$\frac{\min(t_g, t_h)}{2}\,(1 + \mathsf{cost}(g) + \mathsf{cost}(h)) \leq t_f(d_f + 1) - 1. \tag{1}$$

Proposition 1 yields the following remark. Suppose that f is given in *expanded form*, that is, as the sum of its terms. Evaluating f, when $x_1, \ldots, x_n$ are replaced by actual values $k_1, \ldots, k_n \in \mathbb{K}$, amounts then to at most $t_f(d_f + 1) - 1$ arithmetic operations in $\mathbb{K}$. Assume $g \odot h$ is a syntactic factorization of f. Then evaluating both g and h at $k_1, \ldots, k_n$ may provide a speedup factor in the order of $\min(t_g, t_h)/2$. This observation motivates the introduction of the notions introduced in this section.

Syntactic decomposition. Let T be a binary tree whose internal nodes are the operators $+, -, \times$ and whose leaves belong to $\mathbb{K} \cup X$. Let p_T be the polynomial represented by T. We say that T is a *syntactic decomposition* of p_T if either (1), (2) or (3) holds:

(1) T consists of a single node which is p_T,
(2) if T has root $+$ (resp. $-$) with left subtree T_ℓ and right subtree T_r then we have:
- (a) T_ℓ, T_r are syntactic decompositions of two polynomials $p_{T_\ell}, p_{T_r} \in \mathbb{K}[X]$,
- (b) $p_T = p_{T_\ell} \oplus p_{T_r}$ (resp. $p_T = p_{T_\ell} \ominus p_{T_r}$) holds,

(3) if T has root $\times$, with left subtree T_ℓ and right subtree T_r then we have:
- (a) T_ℓ, T_r are syntactic decompositions of two polynomials $p_{T_\ell}, p_{T_r} \in \mathbb{K}[X]$,
- (b) $p_T = p_{T_\ell} \odot p_{T_r}$ holds.

We shall describe an algorithm that computes a syntactic decomposition of a polynomial. The design of this algorithm is guided by our objective of processing polynomials with many terms. Before presenting this algorithm, we make a few observations.

First, suppose that f admits a syntactic factorization $f = g \odot h$. Suppose also that the monomials of g and h are known, but not their coefficients. Then, one can easily deduce the coefficients of both g and h, see Proposition 3 hereafter.

Secondly, suppose that f admits a syntactic factorization $g\,h$ while nothing is known about g and h, except their numbers of terms. Then, one can set up a system of polynomial equations to compute the terms of g and h. For instance with $t_f = 4$ and $t_g = t_h = 2$, let $f = M + N + P + Q$, $g = X + Y$, $h = Z + T$. Up to renaming the terms of f, the following system must have a solution: $XZ = M,\ XT = P,\ YZ = N\ \text{ and }\ YT = Q$.

This implies that $M/P = N/Q$ holds. Then, one can check that $(g, g', M/g, N/g')$ is a solution for (X, Y, Z, T), where $g = \gcd(M, P)$ and $g' = \gcd(N, Q)$.

Thirdly, suppose that f admits a syntactic factorization $f = g \odot h$ while nothing is known about g, h including numbers of terms. In the worst case, all integer pairs (t_g, t_h) satisfying $t_g t_h = t_f$ need to be considered, leading to an algorithm which is exponential in t_f. This approach is too costly for our targeted large polynomials. Finally, in practice, we do not know whether f admits a syntactic factorization or not. Traversing every subset of $\mathsf{terms}(f)$ to test this property would lead to another combinatorial explosion.

3 The Hypergraph Method

Based on the previous observations, we develop the following strategy. Given a set of monomials $\mathcal{M}$, which we call *base monomial set*, we look for a polynomial p such that $\mathsf{terms}(p) \subseteq \mathsf{terms}(f)$, and p admits a syntactic factorization gh w.r.t $\mathcal{M}$. Replacing f by $f - p$ and repeating this construction would eventually produce a *partial syntactic factorization* of f, as defined below. The algorithm $\mathsf{ParSynFactorization}(f, \mathcal{M})$ states this strategy formally. We will discuss the choice and computation of the set $\mathcal{M}$ at the end of this section. The key idea of Algorithm ParSynFactorization is to consider a hypergraph $\mathsf{HG}(f, \mathcal{M})$ which detects "candidate syntactic factorizations".

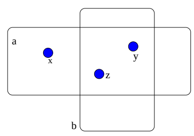
Partial syntactic factorization. A set of pairs $\{(g_1, h_1), (g_2, h_2), \ldots, (g_e, h_e)\}$ of polynomials and a polynomial r in $\mathbb{K}[x_1, \ldots, x_n]$ is a *partial syntactic factorization* of f w.r.t. $\mathcal{M}$ if the following conditions hold:

1. $\forall i = 1 \cdots e$, $\mathsf{mons}(g_i) \subseteq \mathcal{M}$,
2. no monomials in $\mathcal{M}$ divides a monomial of r,
3. $f = (g_1 \odot h_1) \oplus (g_2 \odot h_2) \oplus \cdots \oplus (g_e \odot h_e) \oplus r$ holds.

Assume that the above conditions hold. We say this partial syntactic factorization is *trivial* if each $g_i \odot h_i$ is a trivial syntactic factorization. Observe that all g_i for $1 \le i \le e$ and r do not admit any nontrivial partial syntactic factorization w.r.t. $\mathcal{M}$, whereas it is possible that one of h_i's admits a nontrivial partial syntactic factorization.

Hypergraph $\mathsf{HG}(f, \mathcal{M})$. Given a polynomial f and a set of monomials $\mathcal{M}$, we construct a hypergraph $\mathsf{HG}(f, \mathcal{M})$ as follows. Its vertex set is $\mathcal{V} = \mathcal{M}$ and its hyperedge set $\mathcal{E}$ consists of all nonempty sets $E_q := \{m \in \mathcal{M} \mid m\,q \in \mathsf{mons}(f)\}$, for an arbitrary monomial q. Observe that if a term of f is not the multiple of any monomials in $\mathcal{M}$, then it is not involved in the construction of $\mathsf{HG}(f, \mathcal{M})$. We call such a term *isolated*.

Example. For $f = ay + az + by + bz + ax + aw \in \mathbb{Q}[x, y, z, w, a, b]$ and $\mathcal{M} = \{x, y, z\}$, the hypergraph $\mathsf{HG}(f, \mathcal{M})$ has 3 vertices x, y, z and 2 hyperedges $E_a = \{x, y, z\}$ and $E_b = \{y, z\}$. A partial syntactic factorization of f w.r.t $\mathcal{M}$ consists of $\{(y + z, a + b), (x, a)\}$ and aw.



We observe that a straightforward algorithm computes $\mathsf{HG}(f, \mathcal{M})$ in $O(|\mathcal{M}|\, n\, t)$ bit operations. The following proposition, whose proof is immediate, suggests how $\mathsf{HG}(f, \mathcal{M})$ can be used to compute a partial syntactic factorization of f w.r.t. $\mathcal{M}$.

Proposition 2 *Let $f, g, h \in \mathbb{K}[X]$ such that $f = g \odot h$ and $\mathsf{mons}(g) \subseteq \mathcal{M}$ both hold. Then, the intersection of all E_q, for $q \in \mathsf{mons}(h)$, contains $\mathsf{mons}(g)$.*

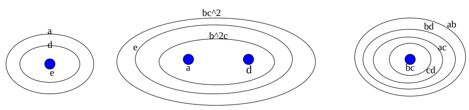
Before stating Algorithm ParSynFactorization, we make a simple observation.

Proposition 3 *Let $F_1, F_2, \ldots, F_c$ be the monomials and $f_1, f_2, \ldots, f_c$ be the coefficients of a polynomial $f \in \mathbb{K}[X]$, such that $f = \sum_{i=1}^{c} f_i F_i$. Let $a, b > 0$ be two integers such that $c = ab$. Given monomials $G_1, G_2, \ldots, G_a$ and $H_1, H_2, \ldots, H_b$ such that the products $G_i H_j$ are all in $\mathsf{mons}(f)$ and are pairwise different. Then, within $O(ab)$ operations in $\mathbb{K}$ and $O(a^2 b^2 n)$ bit operations, one can decide whether $f = g \odot h$, $\mathsf{mons}(g) = \{G_1, G_2, \ldots, G_a\}$ and $\mathsf{mons}(h) = \{H_1, H_2, \ldots, H_b\}$ all hold. Moreover, if such a syntactic factorization exists it can be computed within the same time bound.*

Proof. Define $g = \sum_{i=1}^{a} g_i G_i$ and $h = \sum_{i=1}^{b} h_i H_i$ where $g_1, \ldots, g_a$ and $h_1, \ldots, h_b$ are unknown coefficients. The system to be solved is $g_i h_j = f_{ij}$, for all $i = 1 \cdots a$ and all $j = 1 \cdots b$ where f_{ij} is the coefficient of $G_i H_j$ in p. To set up this system $g_i h_j = f_{ij}$, one needs to locate each monomial $G_i H_j$ in $\mathsf{mons}(f)$. Assuming that each exponent of a monomial is a machine word, any two monomials of $\mathbb{K}[x_1, \ldots, x_n]$ are compared within $O(n)$ bit operations. Hence, each of these ab monomials can be located in $\{F_1, F_2, \ldots, F_c\}$ within $O(cn)$ bit operations and the system is set up within $O(a^2 b^2 n)$ bit operations. We observe that if $f = g \odot h$ holds, one can freely set g_1 to 1 since the coefficients are in a field. This allows us to deduce $h_1, \ldots, h_b$ and then $g_2, \ldots, g_a$ using $a + b - 1$ equations. The remaining equations of the system should be used to check if these values of $h_1, \ldots, h_b$ and $g_2, \ldots, g_a$ lead indeed to a solution. Overall, for each of the ab equations one simply needs to perform one operation in $\mathbb{K}$.

Remark on Algorithm 1. Following the property of the hypergraph $\mathsf{HG}(f, \mathcal{M})$ given by Proposition 2, we use a greedy strategy and search for the largest hyperedge intersection in $\mathsf{HG}(f, \mathcal{M})$. Once such intersection is found, we build a candidate syntactic factorization from it. However, it is possible that the equality in Line 12 does not hold. For example, when $\mathcal{M} = Q = \{a, b\}$, we have $|N| = 3 \neq 2 \times 2 = |M| \cdot |Q|$. When the equality $|N| = |M| \cdot |Q|$ holds, there is still a possibility that the system set up as in the proof of Proposition 3 does not have solutions. For example, when $\mathcal{M} = \{a, b\}, Q = \{c, d\}$ and $p = ac + ad + bc + 2\,bd$. Nevertheless, the termination of the while loop in Line 10 is ensured by the following observation. When $|Q| = 1$, the equality $|N| = |M| \cdot |Q|$ always holds and the system set up as in the proof of Proposition 3 always has a solution. After extracting a syntactic factorization from the hypergraph $\mathsf{HG}(f, \mathcal{M})$, we update the hypergraph by removing all monomials in the set N and keep extracting syntactic factorizations from the hypergraph until no hyperedges remain.

Example. Consider $f = 3ab^2c + 5abc^2 + 2ae + 6b^2cd + 10bc^2d + 4de + s$. Our base monomial set $\mathcal{M}$ is chosen as $\{a, bc, e, d\}$. Following Algorithm 1, we first construct the hypergraph $\mathsf{HG}(f, \mathcal{M})$ w.r.t. which the term s is isolated.



Input : a polynomial f given as a sorted set $\mathsf{terms}(f)$, a monomial set $\mathcal{M}$
Output : a partial syntactic factorization of f w.r.t $\mathcal{M}$

$\mathcal{T} \leftarrow \mathsf{terms}(f)$, $\mathcal{F} \leftarrow \emptyset$;
$r \leftarrow \sum_{t \in \mathcal{I}} t$ where $\mathcal{I} = \{t \in \mathsf{terms}(f) \mid (\forall m \in \mathcal{M})\ m \nmid t\}$;
compute the hypergraph $\mathsf{HG}(f, \mathcal{M}) = (\mathcal{V}, \mathcal{E})$;
while *$\mathcal{E}$ is not empty* **do**
  if *$\mathcal{E}$ contains only one edge E_q* **then** $Q \leftarrow \{q\}$, $M \leftarrow E_q$;
  else
    find q, q' such that $E_q \cap E_{q'}$ has the maximal cardinality;
    $M \leftarrow E_q \cap E_{q'}, Q \leftarrow \emptyset$;
    if $|M| < 1$ **then** find the largest edge E_q, $M \leftarrow E_q, Q \leftarrow \{q\}$;
    else for $E_q \in \mathcal{E}$ **do if** $M \subseteq E_q$ **then** $Q \leftarrow Q \cup \{q\}$;
  while *true* **do**
    $N = \{mq \mid m \in M, q \in Q\}$;
    if $|N| = |M| \cdot |Q|$ **then**
      let p be the polynomial such that $\mathsf{mons}(p) = N$ and $\mathsf{terms}(p) \subseteq \mathcal{T}$;
      if $p = g \odot h$ *with* $\mathsf{mons}(g) = M$ *and* $\mathsf{mons}(h) = Q$ **then**
        compute g, h (Proposition 3); break;
    else randomly choose $q \in Q$, $Q \leftarrow Q \setminus \{q\}$, $M \leftarrow \cap_{q \in Q} E_q$;
  for $E_q \in \mathcal{E}$ **do**
    for $m' \in N$ **do**
      if $q \mid m'$ **then** $E_q \leftarrow E_q \setminus \{m'/q\}$;
    if $E_q = \emptyset$ **then** $\mathcal{E} \leftarrow \mathcal{E} \setminus \{E_q\}$;
  $\mathcal{T} \leftarrow \mathcal{T} \setminus \mathsf{terms}(p)$, $\mathcal{F} \leftarrow \mathcal{F} \cup \{g \odot h\}$;
return $\mathcal{F}, r$

Algorithm 1: ParSynFactorization

The largest edge intersection is $M = \{a, d\} = E_{b^2c} \cap E_{bc^2} \cap E_e$ yielding $Q = \{b^2c, bc^2, e\}$. The set N is $\{mq \mid m \in M, q \in Q\} = \{ab^2c, abc^2, ae, b^2cd, bc^2d, de\}$. The cardinality of N equals the product of the cardinalities of M and of Q. So we keep searching for a polynomial p with N as monomial set and with $\mathsf{terms}(p) \subseteq \mathsf{terms}(f)$. By scanning $\mathsf{terms}(f)$ we obtain $p = 3ab^2c + 5abc^2 + 2ae + 6b^2cd + 10bc^2d + 4de$. Now we look for polynomials g, h with respective monomial sets M, Q and such that $p = g \odot h$ holds. The following equality yields a system of equations whose unknowns are the coefficients of g and h: $(g_1a + g_2d)(h_1b^2c + h_2bc^2 + h_3e) = 3ab^2c + 5abc^2 + 2ae + 6b^2cd + 10bc^2d + 4de$. As described in Proposition 3, we can freely set g_1 to 1 and then use 4 out of the 6 equations to deduce h_1, h_2, h_3, g_2; these computed values must verify the remaining equations for $p = g \odot h$ to hold, which is the case here.

$$\begin{cases} g_1h_1 = 3 \\ g_1h_2 = 5 \\ g_1h_3 = 2 \\ g_2h_1 = 6 \end{cases} \overset{g_1=1}{\Longrightarrow} \begin{cases} g_1 = 1 \\ g_2 = 2 \\ h_1 = 3 \\ h_2 = 5 \\ h_3 = 2 \end{cases} \Rightarrow \begin{cases} g_2h_2 = 10 \\ g_2h_3 = \ 4 \end{cases}$$

Now we have found a syntactic factorization of p. We update each edge in the hypergraph, which, in this example, will make the hypergraph empty. After adding $(a+2d, 3b^2c+5bc^2+2e)$ to $\mathcal{F}$, the algorithm terminates with $\mathcal{F}, s$ as output.

One may notice that in Example 3, $h = 3b^2c + 5bc^2 + 2e$ also admits a nontrivial partial syntactical factorization. Computing it will produce a syntactic decomposition of f. When a polynomial which does not admit any nontrivial partial syntactical factorizations w.r.t $\mathcal{M}$ is hit, for instance, g_i or r in a partial syntactic factorization, we directly convert it to an expression tree. To this end, we assume that there is a procedure $\mathsf{ExpressionTree}(f)$ that outputs an expression tree of a given polynomial f. Algorithm 2, which we give for the only purpose of being precise, states the most straight forward way to implement $\mathsf{ExpressionTree}(f)$. Then, Algorithm 3 formally states how to produce a syntactic decomposition of a given polynomial.

Input : a polynomial f given as $\mathsf{terms}(f) = \{t_1, t_2, \ldots, t_s\}$
Output: an expression tree whose value equals f

if $\sharp\mathsf{terms}(f) = 1$ *say* $f = c \cdot x_1^{d_1} x_2^{d_2} \cdots x_k^{d_k}$ **then**
  for $i \leftarrow 1$ **to** k **do**
    $T_i \leftarrow x_i$;
    for $j \leftarrow 2$ **to** d_i **do**
      $T_{i,\ell} \leftarrow T_i, \mathsf{root}(T_i) \leftarrow \times, T_{i,r} \leftarrow x_i$;
  $T \leftarrow$ empty tree, $\mathsf{root}(T) \leftarrow \times$, $T_\ell \leftarrow c$, $T_r \leftarrow T_1$;
  for $i \leftarrow 2$ **to** k **do**
    $T_\ell \leftarrow T, \mathsf{root}(T) \leftarrow \times, T_r \leftarrow T_i$;
else
  $k \leftarrow s/2, f_1 \leftarrow \sum_{i=1}^{k} t_i, f_2 \leftarrow \sum_{i=k+1}^{s} t_i$;
  $T_1 \leftarrow \mathsf{ExpressionTree}(f_1)$;
  $T_2 \leftarrow \mathsf{ExpressionTree}(f_2)$;
  $\mathsf{root}(T) \leftarrow +, T_\ell \leftarrow T_1, T_r \leftarrow T_2$;

Algorithm 2: ExpressionTree

We have stated all the algorithms that support the construction of a syntactic decomposition except for the computation of the base monomial set $\mathcal{M}$. Note that in Algorithm 1 our main strategy is to keep extracting syntactic factorizations from the hypergraph $\mathsf{HG}(f, \mathcal{M})$. For all the syntactic factorizations $g \odot h$ computed in this manner, we have $\mathsf{mons}(g) \subseteq \mathcal{M}$. Therefore, to discover all the possible syntactic factorizations in $\mathsf{HG}(f, \mathcal{M})$, the base monomial set should be chosen so as to contain all the monomials from which a syntactic factorization may be derived. The most obvious choice is to consider the set G of all non constant gcds of any two distinct terms of f. However, $|G|$ could be quadratic in $\#\mathsf{terms}(f)$, which would be a bottleneck on large polynomials f. Our strategy is to choose for $\mathcal{M}$ as the set of the minimal elements of G for the divisibility relation. A straightforward algorithm computes this set $\mathcal{M}$ within $O(t^4 n)$ operations in $\mathbb{K}$; indeed $|\mathcal{M}|$ fits in $|G| = O(t^2)$. In practice, $\mathcal{M}$ is much smaller than G

Input : a polynomial f given as $\mathsf{terms}(f)$
Output: a syntactic decomposition of f

compute the base monomial set $\mathcal{M}$ for f;
if $\mathcal{M} = \emptyset$ **then** return $\mathsf{ExpressionTree}(f)$;
else
    $\mathcal{F}, r \leftarrow \mathsf{ParSynFactorization}(f, \mathcal{M})$;
    for $i \leftarrow 1$ **to** $|\mathcal{F}|$ **do**
        $(g_i, h_i) \leftarrow \mathcal{F}_i, T_i \leftarrow$ empty tree, $\mathsf{root}(T_i) \leftarrow \times$;
        $T_{i,\ell} \leftarrow \mathsf{ExpressionTree}(g_i)$;
        $T_{i,r} \leftarrow \mathsf{SyntacticDecomposition}(h_i)$;
    $T \leftarrow$ empty tree, $\mathsf{root}(T) \leftarrow +, T_\ell \leftarrow \mathsf{ExpressionTree}(r), T_r \leftarrow T_1$;
    for $i \leftarrow 2$ **to** $|\mathcal{F}|$ **do**
        $T_\ell \leftarrow T, \mathsf{root}(T) \leftarrow +, T_r \leftarrow T_i$;

Algorithm 3: SyntacticDecomposition

(for large dense polynomials, $\mathcal{M} = X$ holds) and this choice is very effective. However, since we aim at manipulating large polynomials, the set G can be so large that its size can be a memory bottleneck when computing $\mathcal{M}$. In [2] we address this question: we propose a divide-and-conquer algorithm which computes $\mathcal{M}$ directly from f without storing the whole set G in memory. In addition, the parallel implementation in `Cilk+` shows linear speed-up on 32 cores for sufficiently large input.

4 Complexity Estimates

Given a polynomial f of t terms with total degree d in $\mathbb{K}[X]$, we analyze the running time for Algorithm 3 to compute a syntactic decomposition of f. Assuming that each exponent in a monomial is encoded by a machine word, each operation (GCD, division) on a pair of monomials of $\mathbb{K}[X]$ requires $O(n)$ bit operations. Due to the different manners of constructing a base monomial set, we keep $\mu := |\mathcal{M}|$ as an input complexity measure. As mentioned in Section 3, $\mathsf{HG}(f, \mathcal{M})$ is constructed within $O(\mu t n)$ bit operations. This hypergraph contains μ vertices and $O(\mu t)$ hyperedges. We first proceed by analyzing Algorithm 1. To do so, we follow its steps.

- The "isolated" polynomial r can be easily computed by testing the divisibility of each term in f w.r.t each monomial in $\mathcal{M}$, i.e. in $O(\mu \cdot t \cdot n)$ bit operations.
- Each hyperedge in $\mathsf{HG}(f, \mathcal{M})$ is a subset of $\mathcal{M}$. The intersection of two hyperedges can then be computed in $\mu \cdot n$ bit operations. Thus we need $O((\mu t)^2 \cdot \mu n) = O(\mu^3 t^2 n)$ bit operations to find the largest intersection M (Line 7).
- If M is empty, we traverse all the hyperedges in $\mathsf{HG}(f, \mathcal{M})$ to find the largest one. This takes no more than $\mu t \cdot \mu n = \mu^2 t n$ bit operations (Line 9).
- If M is not empty, we traverse all the hyperedges in $\mathsf{HG}(f, \mathcal{M})$ to test if M is a subset of it. This takes at most $\mu t \cdot \mu n = \mu^2 t n$ bit operations (Line 10).
- Line 6 to Line 10 takes $O(\mu^3 t^2 n)$ bit operations.

- The set N can be computed in $\mu \cdot \mu t \cdot n$ bit operations (Line 12).
- by Proposition 3, the candidate syntactic factorization can be either computed or rejected in $O(|M|^2 \cdot |Q|^2 n) = O(\mu^4 t^2 n)$ bit operations and $O(\mu^2 t)$ operations in $\mathbb{K}$ (Lines 13 to 16).
- If $|N| \neq |M| \cdot |Q|$ or the candidate syntactic factorization is rejected, we remove one element from Q and repeat the work in Line 12 to Line 16. This while loop ends before or when $|Q| = 1$, hence it iterates at most $|Q|$ times. So the bit operations of the while loop are in $O(\mu^4 t^2 n \cdot \mu t) = O(\mu^5 t^3 n)$ while operations in $\mathbb{K}$ are within $O(\mu^2 t \cdot \mu t) = O(\mu^3 t^2)$ (Line 11 to Line 17).
- We update the hypergraph by removing the monomials in the constructed syntactic factorization. The two nested for loops in Line 18 to Line 21 take $O(|\mathcal{E}| \cdot |N| \cdot n) = O(|\mathcal{E}| \cdot |M| \cdot |Q| \cdot n) = O(\mu t \cdot \mu \cdot \mu t \cdot n) = O(\mu^3 t^2 n)$ bit operations.
- Each time a syntactic factorization is found, at least one monomial in $\mathsf{mons}(f)$ is removed from the hypergraph $\mathsf{HG}(f, \mathcal{M})$. So the while loop from Line 4 to Line 22 would terminate in $O(t)$ iterations.

Overall, Algorithm 1 takes $O(\mu^5 t^4 n)$ bit operations and $O(\mu^3 t^3)$ operations in $\mathbb{K}$. One easily checks from Algorithm 2 that an expression tree can be computed from f (where f has t terms and total degree d) within in $O(ndt)$ bit operations. In the sequel of this section, we analyze Algorithm 3. We make two preliminary observations. First, for the input polynomial f, the cost of computing a base monomial set can be covered by the cost of finding a partial syntactic factorization of f. Secondly, the expression trees of all g_i's (Line 7) and of the isolated polynomial r (Line 9) can be computed within $O(ndt)$ operations. Now, we shall establish an equation that rules the running time of Algorithm 3. Assume that $\mathcal{F}$ in Line 4 contains e syntactic factorizations. For each g_i, h_i such that $(g_i, h_i) \in \mathcal{F}$, let the number of terms in h_i be t_i and the total degree of h_i be d_i. By the specification of the partial syntactic factorization, we have $\sum_{i=1}^{e} t_i \leq t$. It is easy to show that $d_i \leq d - 1$ holds for $1 \leq i \leq e$ as total degree of each g_i is at least 1. We recursively call Algorithm 3 on all h_i's. Let $T_b(t, d, n)(T_{\mathbb{K}}(t, d, n))$ be the number of bit operations (operations in $\mathbb{K}$) performed by Algorithm 3. We have the following recurrence relation,

$$T_b(t, d, n) = \sum_{i=1}^{e} T_b(t_i, d_i, n) + O(\mu^5 t^4 n), T_{\mathbb{K}}(t, d, n) = \sum_{i=1}^{e} T_{\mathbb{K}}(t_i, d_i, n) + O(\mu^3 t^3),$$

from which we derive that $T_b(t, d, n)$ is within $O(\mu^5 t^4 n d)$ and $T_{\mathbb{K}}(t, d, n)$ is within $O(\mu^3 t^3 d)$. Next, one can verify that if the base monomial set $\mathcal{M}$ is chosen as the set of the minimal elements of all the pairwise gcd's of monomials of f, where $\mu = O(t^2)$, then a syntactic decomposition of f can be computed in $O(t^{14} n d)$ bit operations and $O(t^9 d)$ operations in $\mathbb{K}$. If the base monomial set is simply set to be $X = \{x_1, x_2, \ldots, x_n\}$, then a syntactic decomposition of f can be found in $O(t^4 n^6 d)$ bit operations and $O(t^3 n^3 d)$ operations in $\mathbb{K}$.

5 Experimental Results

In this section we discuss the performances of different software tools for reducing the evaluation cost of large polynomials. These tools are based respectively on a multivariate Horner's scheme [3], the `optimize` function with `tryhard` option provided by the computer algebra system Maple and our algorithm presented in Section 3. As described in the introduction, we use the evaluation of resultants of generic polynomials as a driving example. We have implemented our algorithm in the `Cilk++` programming language. We report on different performance measures of our optimized DAG representations as well as those obtained with the other software tools.

Evaluation cost. Figure 1 shows the total number of internal nodes of a DAG representing the resultant $R(a,b)$ of two generic polynomials $a = a_m x^m + \cdots + a_0$ and $b = b_n x^n + \cdots + b_0$ of degrees m and n, after optimizing this DAG by different approaches. The number of internal nodes of this DAG measures the cost of evaluating $R(a,b)$ after specializing the variables $a_m, \ldots, a_0, b_n \ldots, b_0$. The first two columns of Figure 1 gives m and n. The third column indicates the number of monomials appearing in $R(a,b)$. The number of internal nodes of the input DAG, as computed by MAPLE, is given by the fourth column (Input). The fifth column (Horner) is the evaluation cost (number of internal nodes) of the DAG after MAPLE's multivariate Horner's rule is applied. The sixth column (tryhard) records the evaluation cost after MAPLE's optimize function (with the tryhard option) is applied. The last two columns reports the evaluation cost of the DAG computed by our *hypergraph method* (HG) before and after removing common subexpressions. Indeed, our hypergraph method requires this post-processing (for which we use standard techniques running in time linear w.r.t. input size) to produce better results. We note that the evaluation cost of the DAG returned by HG + CSE is less than the ones obtained with the Horner's rule and MAPLE's `optimize` functions.

m	n	#Mon	Input	Horner	tryhard	HG	HG + CSE
4	4	219	1876	977	721	899	549
5	4	549	5199	2673	1496	2211	1263
5	5	1696	18185	7779	4056	7134	3543
6	4	1233	13221	6539	3230	4853	2547
6	5	4605	54269	22779	10678	18861	8432
6	6	14869	190890	69909	31760	63492	24701
7	4	2562	30438	14948	6707	9862	4905
7	5	11380	146988	61399	27363	45546	19148
7	6	43166	601633	219341	-	179870	65770

Fig. 1. Cost to evaluate a DAG by different approaches

Figure 2 shows the timing in seconds that each approach takes to optimize the DAGs analyzed in Figure 1. The first three columns of Figure 2 have the same meaning as in Figure 1. The columns (Horner), (tryhard) show the timing of optimizing these DAGs. The last column (HG) shows the timing to produce the syntactic decompositions with our `Cilk++` implementation on multicores using 1, 4, 8 and 16 cores. All the sequential benchmarks (Horner, tryhard) were conducted on a 64bit Intel Pentium VI Quad CPU 2.40 GHZ machine with 4 MB L2 cache and 3 GB main memory. The parallel benchmarks were carried out on a 16-core machine at SHARCNET (www.sharcnet.ca) with 128 GB memory in total and 8×4096 KB of L2 cache (each integrated by 2 cores). All the processors are Intel Xeon E7340 @ 2.40GHz.

As the input size grows, the timing of the MAPLE `Optimize` command (with tryhard option) grows dramatically and takes more than 40 hours to optimize the resultant of two generic polynomials with degrees 6 and 6. For the generic polynomials with degree 7 and 6, it does not terminate after 5 days. For the largest input (7, 6), our algorithm completes within 5 minutes on one core. Our preliminary implementation shows a speedup around 8 when 16 cores are available. The parallelization of our algorithm is still work in progress (for instance, in the current implementation Algorithm 3 has **not** been parallelized yet). We are further improving the implementation and leave for a future paper reporting the parallelization of our algorithms.

m	n	#Mon	Horner	tryhard	HG (# cores $= 1, 4, 8, 16$)			
4	4	219	0.116	7.776	0.017	0.019	0.020	0.023
5	4	549	0.332	49.207	0.092	0.073	0.068	0.067
5	5	1696	1.276	868.118	0.499	0.344	0.280	0.250
6	4	1233	0.988	363.970	0.383	0.249	0.213	0.188
6	5	4605	4.868	8658.037	3.267	1.477	1.103	0.940
6	6	14869	24.378	145602.915	29.130	9.946	6.568	4.712
7	4	2562	4.377	1459.343	1.418	0.745	0.603	0.477
7	5	11380	24.305	98225.730	22.101	7.687	5.106	3.680
7	6	43166	108.035	>136 hours	273.963	82.497	49.067	31.722

Fig. 2. timing to optimize large polynomials

Evaluation schedule. Let T be a syntactic decomposition of an input polynomial f. Targeting multi-core evaluation, our objective is to decompose T into p sub-DAGs, given a fixed parameter p, the number of available processors. Ideally, we want these sub-DAGs to be balanced in size such that the "span" of the intended parallel evaluation can be minimized. These sub-DAGs should also be independent to each other in the sense that the evaluation of one does not depend on the evaluation of another. In this manner, these sub-DAGs can be assigned to different processors. When p processors are available, we call "p-schedule" such a decomposition. We report on the 4 and 8-schedules generated from our syntactic decompositions. The column "T" records the size of a syntactic decomposition, counting the number of nodes. The column "$\#$CS" indicates the number of common subexpressions. We notice that the amount of work assigned to each sub-DAG is balanced. However, scheduling the evaluation of the common subexpressions is still work in progress.

Benchmarking generated code. We generated 4-schedules of our syntactic decompositions and compared with three other methods for evaluating our test polynomials on a large number of uniformly generated random points over Z/pZ where $p = 2147483647$ is the largest 31-bit prime number. Our experimental data are summarized in Figure 4. Out the four different evaluation methods, the first three are sequential and are based on the following DAGs: the original MAPLE DAG (labeled as Input), the DAG computed by our hypergraph method (labeled as HG), the HG DAG further optimized by CSE

m	n	T	#CS	4-schedule	8-schedule
6	5	8432	1385	1782, 1739, 1760, 1757	836, 889, 884, 881, 892, 886, 886, 869
6	6	24701	4388	4939, 5114, 5063, 5194	2436, 2498, 2496, 2606, 2535, 2615, 2552, 2555
7	5	19148	3058	3900, 4045, 4106, 4054	1999, 2049, 2078, 1904, 2044, 2019, 1974, 2020
7	6	65770	10958	13644, 13253, 14233, 13705	6710, 6449, 7117, 6802, 6938, 7025, 6807, 6968

Fig. 3. parallel evaluation schedule

(labeled as HG + CSE). The last method uses the 4-schedule generated from the DAG obtained by HG + CSE. All these evaluation schemes are automatically generated as a list of SLPs. When an SLP is generated as one procedure in a source file, the file size grows linearly with the number of lines in this SLP. We observe that gcc 4.2.4 failed to compile the resultant of generic polynomials of degree 6 and 6 (the optimization level is 2). In Figure 4, we report the timings of the four approaches to evaluate the input at $10K$ and $100K$ points. The first four data rows report timings where the gcc optimization level is 0 during the compilation, and the last row shows the timings with the optimization at level 2. We observe that the optimization level affects the evaluation time by a factor of 2, for each of the four methods. Among the four methods, the 4-schedule method is the fastest and it is about 20 times faster than the first method.

m	n	#point	Input	HG	HG+CSE	4-schedule	#point	Input	HG	HG+CSE	4-schedule
6	5	10K	14.490	2.675	1.816	0.997	100K	144.838	26.681	18.103	9.343
6	6	10K	57.853	18.618	4.281	2.851	100K	577.624	185.883	42.788	28.716
7	5	10K	46.180	11.423	4.053	2.104	100K	461.981	114.026	40.526	19.560
7	6	10K	190.397	54.552	13.896	8.479	100K	1902.813	545.569	138.656	81.270
6	5	10K	6.611	1.241	0.836	0.435	100K	66.043	12.377	8.426	4.358

Fig. 4. timing to evaluate large polynomials

References

1. Melvin A. Breuer. Generation of optimal code for expressions via factorization. *Commun. ACM*, 12(6):333–340, 1969.
2. C. E. Leiserson, L. Li, M. Moreno Maza, and Y. Xie. Parallel computation of the minimal elements of a poset. In *Proc. PASCO'10*. ACM Press, 2010.
3. J. Carnicer and M. Gasca. Evaluation of multivariate polynomials and their derivatives. *Mathematics of Computation*, 54(189):231–243, 1990.
4. M. Ceberio and V. Kreinovich. Greedy algorithms for optimizing multivariate horner schemes. *SIGSAM Bull.*, 38(1):8–15, 2004.
5. Intel Corporation. Cilk++. http://www.cilk.com/.
6. J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge Univ. Press, 1999.
7. A. Hosangadi, F. Fallah, and R. Kastner. Factoring and eliminating common subexpressions in polynomial expressions. In *ICCAD'04* , pages 169–174, 2004. IEEE Computer Society.
8. J. M. Peña. On the multivariate Horner scheme. *SIAM J. Numer. Anal.*, 37(4):1186–1197, 2000.
9. J. M. Peña and Thomas Sauer. On the multivariate Horner scheme ii: running error analysis. *Computing*, 65(4):313–322, 2000.