Efficient Evaluation of Large Polynomials

(Spine title: Efficient Evaluation of Large Polynomials)

(Thesis format: Monograph)

by

Liyun Li

Graduate Program
in
Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

School of Graduate and Postdoctoral Studies Studies
The University of Western Ontario
London, Ontario, Canada

© L. Li 2010

THE UNIVERSITY OF WESTERN ONTARIO
THE SCHOOL OF GRADUATE AND POSTDOCTORAL STUDIES

CERTIFICATE OF EXAMINATION

Supervisor:

Dr. Marc Moreno Maza

Examination committee:

Dr. David Jeffrey

Dr. Hanan Lutfiyya

Dr. Stephen Watt

The thesis by

Liyun Li

entitled:

Efficient Evaluation of Large Polynomials

is accepted in partial fulfillment of the
requirements for the degree of
Master of Science

Date ______________________

____
Chair of the Thesis Examination Board

Abstract

In scientific computing, it is often required to evaluate a polynomial expression (or a matrix depending on some variables) at many points which are not known in advance or with coordinates containing "symbolic expressions". In these circumstances, standard evaluation schemes, such as those based on Fast Fourier Transforms do not apply.

Given a polynomial f expressed as the sum of its terms, we propose an algorithm which generates a representation of f optimizing the process of evaluating f at some points. In addition, this evaluation of f can be done efficiently in terms of data locality and parallelism.

We have implemented our algorithm in the Cilk++ concurrency platform and our implementation achieves nearly linear speedup on 16 cores with large enough input. For some large polynomials, the generated schedule can be evaluated at least 10 times faster than the schedules produced by other available software solutions. Moreover, our code can handle much larger input polynomials.

Keywords. Polynomial evaluation, Parallel evaluation schedule generation, Parallelism, Data locality, Straight-line program.

Acknowledgments

It is a pleasure to convey my gratitude to many people who contributed in assorted ways to my project.

I am heartily thankful to my supervisor, Marc Moreno Maza, whose encouragement, enthusiasm and guidance enabled me to develop an understanding of the subject. This thesis would not have been possible without his support from the initial to the final level.

Great thanks to my colleagues Yuzhen Xie, Xin Li, Changbo Chen, Wei Pan, Sardar Anisul Haque, Paul Vrbik, Rong Xiao for providing a stimulating and fun environment in which to learn and grow. Many thanks go in particular to Bill Naylor, who kindly grants me his time for valuable discussion on this project. I would like to show my gratitude to the Cilk++ development group (at CilkArts and then at Intel Corp). I have benefited by their advice and discussion on their forum. I am also grateful to our colleagues at *Maplesoft*, in particular Jürgen Gerhard and Clare So, for our cooperation on the `RegularChains` and `Modpn` libraries.

I am thankful to the examination committee, David Jeffrey, Hanan Lutfiyya and Stephen Watt for their precious time.

For this research project, I was partially supported by the Shared Hierarchical Academic Research Computing Network (SHARCNET) and the MITACS full project *Mathematics of Computer Algebra and Analysis* (MOCAA). The benchmarks were also made possible by the dedicated resource program of SHARCNET.

I wish to thank my entire extended family, my grandmother, grandmother-in-law, my parents, parents-in-law, my sisters, siblings-in-law, my husband, my niece, nephews, and my nephew-in-law for providing a loving environment for me. To them I dedicate this thesis.

Lastly, I wish to thank my husband, Wei Pan. Everything is in no words.

Contents

Certificate of Examination **ii**

Abstract **iii**

Acknowledgments **iv**

Table of Contents **v**

List of Algorithms **viii**

List of Figures **x**

List of Tables **xi**

1 Introduction **1**
- 1.1 Main Results 6
  - 1.1.1 Syntactic Decomposition 6
  - 1.1.2 Parallelization of the Hypergraph Method 8
  - 1.1.3 Evaluation Schedule and Experimental Results 9
- 1.2 Organization of the Thesis 9

2 Background and Literature Review **10**
- 2.1 Canonical Representation of Multivariate Polynomials 10
  - 2.1.1 Recursive Representation 11
  - 2.1.2 Sparse Distributed Representation 11
- 2.2 Straight-Line Programs 12
  - 2.2.1 Related Work on Straight-Line Program Representation 14
  - 2.2.2 Implementation of Straight-Line Programs 15
- 2.3 Optimizing the Evaluation of Polynomials 16
  - 2.3.1 Common Subexpression Elimination 16

2.3.2 Horner Scheme 19
2.3.3 Algorithms Based on Combinatorial Structures 20
2.4 Parallel Evaluation of DAGs 20
2.5 Other Related Work 23

3 Syntactic Decomposition 24

4 The Hypergraph Method 29
4.1 Partial Syntactic Factorization 29
4.2 Base Monomial Set 37
4.3 Syntactic Decomposition 38

5 Complexity Estimates 41
5.1 Monomial Set Construction 41
5.2 Hypergraph Construction 41
5.3 Computing Partial Syntactic Factorization 42
5.4 Expression Tree Construction 43
5.5 Computing Syntactic Decomposition 43

6 Implementation and Parallelization of the Hypergraph Method 45
6.1 Data Structures 45
6.1.1 Monomials, Terms and Polynomials 46
6.1.2 Hypergraphs 46
6.1.3 Syntactic Decomposition 47
6.2 Parallelization 49
6.2.1 Merging Two Sets of Minimal Elements 50
6.2.2 Computation of Base Monomial Set 53
6.2.3 Construction of the Hypergraph 58
6.2.4 Computation of the Largest Intersection of Hyperedges 60
6.2.5 Identification of Largest Hyperedge 61
6.2.6 Collecting Hyperedges Including the Largest Intersection . . . 61
6.2.7 Multiplication of Two Sets of Monomials 62
6.2.8 Solving Coefficients of a Candidate Syntactic Factorization . . 65
6.2.9 Updating Hypergraph by a Syntactic Factorization 66
6.2.10 Syntactic Decomposition 66

7 Evaluation Scheduling 69

8 Experimentation **75**
8.1 Evaluation Cost . . . 75
8.2 Timing to Optimize Large Polynomials . . . 77
8.3 Evaluation Schedule . . . 80

9 Parallel Computation of the Minimal Elements of a Partially Ordered Set **83**
9.1 The Algorithm . . . 85
9.2 Complexity Analysis and Experimentation . . . 87
9.3 Base Monomial Set Computation . . . 91
9.4 Transversal Hypergraph Generation . . . 92
9.5 Some Remarks . . . 99

10 Conclusions and Future Work **101**

Appendices **104**

A An Example Illustrating the Typical Output of Different Optimization Techniques for a Polynomial Expression **104**

Bibliography **107**

Curriculum Vita **113**

List of Algorithms

1 CommonSubexpressionElimination 18

2 NaiveConstructHypergraph 31
3 ParSynFactorization 34
4 NaiveBaseMonomialSet 38
5 ExpressionTree 39
6 SyntacticDecomposition 40

7 SerialMinMerge 51
8 ParallelMinMerge 52
9 ParallelBaseMonomials 56
10 SelfBaseMonomials 56
11 CrossBaseMonomials 56
12 HalfCrossBaseMonomials 57
13 SerialCrossBaseMonomials 57
14 ParallelConstructHypergraph 59
15 NaiveIntersection 60
16 ParallelIntersection 60
17 ParallelLargestEdge 61
18 ParallelSuperSet 62
19 NaiveMulMonomials 63
20 MulMonomials 63
21 MulMerge 64
22 NaiveMulMerge 65
23 Update 67
24 NaiveUpdate 67

25 ScheduleBinaryTree 73
26 MaximalTreeRoots 73

27 SerialMinPoset 86
28 ParallelMinPoset 86
29 ParallelTransversal 93
30 ParallelHypMerge 94
31 HalfParallelHypMerge 94

List of Figures

1.1 A hypergraph . . . 7

2.1 Input DAG of MM . . . 21
2.2 DAG after applying MM . . . 21
2.3 Input DAG of *shunt* . . . 22
2.4 DAG after applying *shunt* . . . 22

4.1 Computation process of partial syntactic factorization . . . 33

6.1 Illustration of Algorithm 8 ParallelMinMerge . . . 52
6.2 Illustration of Algorithm 9 ParallelBaseMonomials . . . 54
6.3 Illustration of Algorithm 10 SelfBaseMonomials . . . 54
6.4 Illustration of Algorithm 11 CrossBaseMonomials . . . 55
6.5 Illustration of Algorithm 12 HalfCrossBaseMonomials . . . 55

7.1 Evaluate a DAG in two steps . . . 70
7.2 The load of nodes in a binary tree. . . . 71
7.3 A portion of a binary tree with 615 nodes. Each shaded node is the root of a maximal subtree, with children omitted. . . . 73

8.1 Scalability analysis for parallelized SyntacticDecomposition by Cilkview . . 79

9.1 Scalability analysis for ParallelMinPoset by Cilkview . . . 90
9.2 Scalability analysis for ParallelBaseMonomials by Cilkview . . . 91
9.3 Scalability analysis on ParallelTransversal for data mining problems by Cilkview 96
9.4 Scalability analysis on ParallelTransversal for K_{40}^{5} and K_{30}^{7} by Cilkview . . 98

List of Tables

8.1 Cost to evaluate resultants by different approaches 76
8.2 Cost to evaluation hand made input polynomials 77
8.3 Timing to optimize large polynomials 78
8.4 Speedup of parallel SyntacticDecomposition 80
8.5 Parallel evaluation 4-schedule 81
8.6 Parallel evaluation 8-schedule 81
8.7 Timing to evaluate large polynomials at 10K Points 81
8.8 Timing to evaluate large polynomials at 100K Points 82

9.1 Tests for examples from [39] 97
9.2 Tests for the Kuratowski hypergraphs 99
9.3 Tests for the Lovasz hypergraphs 99

Chapter 1

Introduction

This thesis is dedicated to the efficient evaluation of large polynomials. In scientific computing, it is frequently required to evaluate a polynomial (or a matrix depending on some variables) at many points which are not known in advance and whose coordinates may contain "symbolic expressions". For instance, in the *Hensel-Newton lifting techniques* [37], which are used in many places, a key computation step is to evaluate polynomials.

We review Newton's root finding method as an example of an algorithm where a polynomial needs to be evaluated at many points. This well known algorithm finds successively better approximations of a root of a given polynomial (or a more general function) f. The process starts with an initial guess value x_0 and the sequence of approximations are computed by

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

This iteration continues until reaching a value x_n such that $f(x_n)$ satisfies a specified tolerance value. In this process, we evaluate f and f' at a list of inputs $x_0, x_1, \ldots, x_n$. Except the initial guess x_0, the value of each x_i for $i = 1, \ldots, n$ is computed from the previous approximation x_{i-1}. Thus their values are not known in advance. If we are provided with a representation of f which permits a cheaper evaluation, then all the evaluations can be performed with less cost. In cases like this, it is worth the effort computing a representation of f making its evaluation more efficient.

We observe that when the evaluation points are known in advance specific techniques are available, for instance those based on Fast Fourier Transforms (FFTs) or subproduct trees. We refer to Chapters 8 and 10 in [37] for these techniques. We note also that when the evaluation points are numerical values and when the poly-

nomials are of moderate size, the evaluation process is likely not to be a bottleneck in the targeted computation. However, in this thesis, we are concerned with the situation where the polynomials to be evaluated are potentially large and where the evaluation points are not necessarily numerical. This scenario is typical in symbolic computation.

The initial motivation of this study arose from the computation of polynomial resultants, which essentially play the role for non-linear systems that determinants play for linear systems, that is, they provide conditions for a given system to have solutions. The resultant of two univariate polynomials can be defined as the determinant of the so-called *Sylvester matrix*. Let

$$a = a_m x^m + \cdots + a_1 x + a_0 \quad \text{and} \quad b = b_n x^n + \cdots + b_1 x + b_0$$

be two *generic* univariate polynomials of respective positive degrees m and n. By generic polynomials, we mean that $a_m, \ldots, a_1, a_0, b_n, \ldots, b_1, b_0$ are treated as independent symbols. The Sylvester matrix $\mathrm{Sylv}(a, b, x)$ of a and b is the square matrix of order $n + m$ defined as

$$\mathrm{Sylv}(a,b,x) = \begin{bmatrix} a_m & & & & b_n & & & & \\ a_{m-1} & a_m & & & b_{n-1} & b_n & & & \\ \vdots & \vdots & \ddots & & \vdots & \vdots & \ddots & & \\ \vdots & \vdots & & a_m & b_1 & \vdots & & \ddots & \\ \vdots & \vdots & & a_{m-1} & b_0 & \vdots & & & \ddots \\ \vdots & \vdots & & \vdots & & b_0 & & & b_n \\ a_0 & \vdots & & \vdots & & & \ddots & & b_{n-1} \\ & a_0 & & \vdots & & & & \ddots & \vdots \\ & & \ddots & \vdots & & & & \ddots & \vdots \\ & & & a_0 & & & & & b_0 \end{bmatrix}.$$

Then the resultant of a and b in x is $\mathrm{Det}(\mathrm{Sylv}(\mathrm{a}, \mathrm{b}, \mathrm{x}))$. We simply denote it by $R(a, b)$. Here are a few examples. For $m = n = 1$, the Sylvester matrix of a and b is

$$\mathrm{Sylv}(a,b,x) = \begin{bmatrix} a_1 & b_1 \\ a_0 & b_0 \end{bmatrix}$$

and we have:

$$R(a,b) = a_1 b_0 - a_0 b_1.$$

For $m = n = 2$, the Sylvester matrix of a and b is

$$\mathrm{Sylv}(a, b, x) = \begin{bmatrix} a_2 & 0 & b_2 & 0 \\ a_1 & a_2 & b_1 & b_2 \\ a_0 & a_1 & b_0 & b_1 \\ 0 & a_0 & 0 & b_0 \end{bmatrix}$$

and we have:

$$R(a, b) = a_0^2\, b_2^2 - a_0\, a_1\, b_1\, b_2 - 2\, a_0\, a_2\, b_0\, b_2 + a_0\, a_2\, b_1^2 + a_1^2\, b_0\, b_2 - a_1\, a_2\, b_0\, b_1 + a_2^2\, b_0^2.$$

Suppose that $a_m, \ldots, a_1, a_0, b_n, \ldots, b_1, b_0$ are specialized to (i.e. replaced by) some polynomials $\alpha_m, \ldots, \alpha_1, \alpha_0, \beta_n, \ldots, \beta_1, \beta_0$ in some other variables $c_1, \ldots, c_q$. Let us denote by $R(\alpha, \beta)$ the "specialized" resultant. If these α_i's and β_j's are large in size, then computing $R(\alpha, \beta)$ as a polynomial in $c_1, \ldots, c_q$, expressed as the sum of its terms, may become practically impossible. However, if $R(a, b)$ was originally represented as a straight-line program (SLP for short) with $a_m, \ldots, a_1, a_0, b_n, \ldots, b_1, b_0$ as input nodes and if the α_i's and β_j's are also represented as SLPs with $c_1, \ldots, c_q$ as input nodes, then it is still possible to manipulate $R(\alpha, \beta)$. To illustrate this challenge, let us consider two simple examples.

Example 1. *Assume $m = n = 1$. Recall that in this case, we have:*

$$R(a, b) = a_1 b_0 - a_0 b_1.$$

Let us specialize a_1, a_0, b_1, b_0 to generic bivariate polynomials in variables u, v, with coefficients $e_0, \ldots, e_{11}$:

$\alpha_1 := e_0 u + e_1 v + e_2$, $\alpha_0 := e_3 u + e_4 v + e_5$, $\beta_1 := e_6 u + e_7 v + e_8$ and $\beta_0 := e_9 u + e_{10} v + e_{11}$.

Replacing a_1, a_0, b_1, b_0 by α_1, α_0, β_1, β_0 respectively in $R(a, b)$ and expanding the result, we obtain the following expression:

$$\begin{aligned} R(\alpha, \beta) \;=\; & e_0\, e_9\, u^2 + e_0\, e_{10}\, uv + e_1\, e_9\, uv + e_1\, e_{10}\, v^2 - e_3\, e_6\, u^2 - \\ & e_3\, e_7\, uv - e_4\, e_6\, uv - e_4\, e_7\, v^2 + e_0\, e_{11}\, u + e_1\, e_{11}\, v + \\ & e_2\, e_9\, u + e_2\, e_{10}\, v - e_3\, e_8\, u - e_4\, e_8\, v - e_5\, e_6\, u - \\ & e_5\, e_7\, v + e_2\, e_{11} - e_5\, e_8 \end{aligned}$$

The above expression has 18 terms, each of them has degree 2, 3 or 4. If

$u, v, e_0, \ldots, e_{11}$ are all specialized to numerical values, evaluating the above expression (processing one term after another) will amount to 42 multiplications and 17 additions. Meanwhile, the following straight-line program representation of $R(\alpha, \beta)$ has 19 instructions, which implies that $R(\alpha, \beta)$ can be computed within 19 arithmetic operations (additions, multiplications).

$$
\begin{array}{lcl}
t_1 & \leftarrow & e_0 u \\
t_2 & \leftarrow & e_1 v \\
t_1 & \leftarrow & t_1 + t_2 \\
a_1 & \leftarrow & t_1 + e_2 \\
t_1 & \leftarrow & e_3 u \\
t_2 & \leftarrow & e_4 v \\
t_1 & \leftarrow & t_1 + t_2 \\
a_0 & \leftarrow & t_1 + e_5 \\
t_1 & \leftarrow & e_6 u \\
t_2 & \leftarrow & e_7 v \\
t_1 & \leftarrow & t_1 + t_2 \\
b_1 & \leftarrow & t_1 + e_8 \\
t_1 & \leftarrow & e_9 u \\
t_2 & \leftarrow & e_{10} v \\
t_1 & \leftarrow & t_1 + t_2 \\
b_0 & \leftarrow & t_1 + e_{11} \\
t_1 & \leftarrow & a_1 b_0 \\
t_2 & \leftarrow & a_0 b_1 \\
t_1 & \leftarrow & t_1 - t_2
\end{array}
$$

Example 2. *Assume now that $n = m$ holds and that each of the α_i's and β_j's is a univariate polynomial of degree d, for which each coefficient is an integer which can be stored in at most B bits. From the shape of the Sylvester matrix, it is easy to see that $R(a, b)$ is a multivariate polynomial in $2(n+1)$ variables and of total degree $2(n+1)$. Moreover, once expanded (i.e. written as the sum of its terms) the polynomial $R(a, b)$ has $O(2^n n^n)$ terms. Therefore, the specialized polynomial $R(\alpha, \beta)$ is a univariate polynomial of degree $O(nd)$, where each coefficient fits within $O(2nB + (d+1)^{2n} + 2^n n^n)$ bits. This latter estimate is very rough and pessimistic. However, if the α_i's and β_j's are sufficiently dense and generic, the degree estimate $O(nd)$ is sharp and most coefficients will occupy $\Omega(nB)$ bits. Thus, in this case, the space requirement for storing $R(\alpha, \beta)$ is expected to be $\Omega(n^2 dB)$ bits.*

In contrast, if $R(a,b)$ and each of each of the α_i's , β_j's are encoded as SLPs, then the total space requirement for $R(\alpha,\beta)$ is simply $O(n^4\log(n) + ndB)$ bits. Indeed, for computing a SLP representation of $R(a,b)$, one can use the techniques of Llovet, Castaño and Martínez [45] (based on Jounaidi Abdeljaoued's version [1] of Stuart J. Berkowitz's algorithm [5]) which leads the estimate $O(n^4\log(n))$. The estimate $O(ndB)$ is obtained by applying Horner's rule to each of the polynomials α_i's , β_j's.

Since in practice n is small, say $2 \leq n \leq 10$, it is likely to be much smaller than d and B. This suggests that the SLP representation is much more efficient for evaluating $R(\alpha,\beta)$ than the expanded form.

The previous discussion about the specialization of generic polynomial resultants suggests that, for the purpose of evaluating a polynomial $F(x_1,\ldots,x_n)$, one should look for polynomials $G(y_1,\ldots,y_p), H_1(x_1,\ldots,x_n),\ldots,H_p(x_1,\ldots,x_n)$ satisfying

$$F(x_1,\ldots,x_n) = G(H_1(x_1,\ldots,x_n),\ldots,H_p(x_1,\ldots,x_n)).$$

This *functional decomposition problem* is a classical topic in computer algebra and its applications [18, 21, 26, 27]. For multivariate polynomials, this is a computationally hard problem. Moreover, for the case of large (and generally random) polynomials that we aim at processing, these techniques are not suitable.

We stress the fact that the techniques presented in this thesis do not make any assumptions about the input polynomials and that they are designed to process very large objects. We simply use the example of *resultants of generic polynomials* as an illustrative well-known problem from computer algebra.

Before summarizing the results of this thesis, we discuss the case of univariate polynomials, for which the question of evaluation (as well as that of functional decomposition) is well understood. In fact, it has been shown that *Horner's rule* is optimal for evaluating arbitrary univariate polynomials (Chapter 1 of [12]). Given an arbitrary univariate polynomial of degree n in variable x,

$$f = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n,$$

Horner's rule transforms f into a sequence of nested additions and multiplications

$$f = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_nx))).$$

Evaluating f with this representation amounts to n additions and n multiplications which matches the lower bound given in [12]. In the case of multivariate polynomials,

there are no known methods proven to be optimal. Some related work on this topic will be reviewed in Chapter 2.

1.1 Main Results

1.1.1 Syntactic Decomposition

The first main result of this work contributes to the reduction on the evaluation cost of polynomials. We propose an algorithm that, for a polynomial given as the sum its terms, computes a so-called *syntactic decomposition* of it which permits a cheaper evaluation of this polynomial. This syntactic decomposition is defined as a binary tree where all operations are *syntactic operations*. By syntactic operation, we mean an arithmetic operation (addition, multiplication) on polynomials which can be performed without combination (or grouping) of terms. See Definition 6 p. 25 for a formal definition. For instance, the expression

$$(a + 2d)(bc(3b + 5c) + 2e) + s$$

is a syntactic decomposition of the polynomial

$$3ab^2c + 5abc^2 + 2ae + 6b^2cd + 10bc^2d + 4de + s,$$

whereas $(x - 1)(x^n + x^{n-1} + \cdots + 1)$ is not a syntactic decomposition of $x^n - 1$ since cancellation terms are needed there. Appendix A (p. 104) gives a more complicated example of syntactic decomposition. By imposing the concept of *syntactic operation*, the evaluation cost of our syntactic decomposition is ensured to never grow comparing to the direct evaluation of the input polynomial. For instance, in the above example, evaluating the syntactic decomposition requires 4 additions and 7 multiplications, whereas the direct evaluation of the input polynomial would cost 6 additions and 20 multiplications.

The algorithm to compute syntactic decompositions will be introduced in Chapter 4. Our algorithm keeps extracting the *syntactic factorization*s from the input polynomial, until no nontrivial syntactic factorizations exist. (Syntactic factorization is one of the syntactic operations, formally defined in Definition 7 page 25.) The key idea of our algorithm is to consider a hypergraph which detects "candidate syntactic factorizations". The construction of this hypergraph is presented by Algorithm 2 in Chapter 4. For the above example $f = 3ab^2c + 5abc^2 + 2ae + 6b^2cd + 10bc^2d + 4de + s$,

the hypergraph, guiding the computation of its syntactic decomposition, is shown in Figure 1.1. In this hypergraph, the set of vertices is $\{a, d, bc, e\}$ and each hyperedge

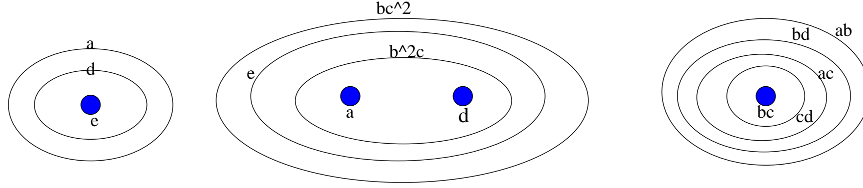


Figure 1.1: A hypergraph

is a collection of vertices defined by a monomial (defining monomial). For example, the hyperedge defined by b^2c contains vertices $\{a, d\}$. Each intersection of hyperedges potentially leads to a "candidate syntactic factorization". For example, the intersection $\{a, d\}$ of the hyperedges defined by the monomials b^2c, bc^2 and e leads to the following syntactic factorization

$$(a + 2d)\,(3b^2c + 5bc^2 + 2e).$$

The monomials of the above syntactic factorization all come from this intersection and the defining monomials of the hyperedges. The coefficients of this syntactic factorization can be derived easily from a system of equations (Proposition 6 p. 32). After extracting this factorization, we update the hypergraph by removing all vertices in each hyperedge related to monomials in this syntactic factorization (Algorithm 3 p. 34). The hypergraph eventually becomes empty and we output a so-called *partial syntactic factorization* of f, which is

$$f = (a + 2d)\,(3b^2c + 5bc^2 + 2e) + s.$$

Recursively applying the same procedure to $3b^2c + 5bc^2 + 2e$, a factor of the parent syntactic factorization, one can compute a syntactic decomposition of f as

$$f = (a + 2d)(bc(3b + 5c) + 2e) + s.$$

As shown in Chapter 5, this algorithm, that we call the *Hypergraph Method*, runs in polynomial time w.r.t. the number of variables, the total degree and the number of terms of the input polynomial in its distributed dense representation.

Our complexity estimates rely on a worst case scenario and our analysis is quite rough due the complicated structure of the main algorithm and its subroutines. Obtaining finer complexity results is work-in-progress. Nevertheless, up to our knowl-

edge, the only algorithm with similar specifications that we could find in the literature [35] has a worst case whose running time is exponential in the number of terms of the input polynomials.

We have implemented our algorithm in the `Cilk++` concurrency platform and our implementation achieves near linear speedup on 16 cores with large enough input. This work is reported in a paper [42] co-authored with Charles E. Leiserson, Marc Moreno Maza and Yuzhen Xie.

1.1.2 Parallelization of the Hypergraph Method

Our purpose is to process very large input polynomials. Thus, it is desirable to parallelize our algorithm for computing syntactic decompositions. This is quite challenging since one of the two core routines (Algorithm 3 p. 34) is inherently sequential. Indeed, one cannot extract two syntactic factorizations concurrently. To overcome this difficulty, one needs to extract parallelism from all subroutines of Algorithm 3 including at a fine-grained level, see Chapter 6. For sufficiently large input polynomials, our `Cilk++` code reaches a nearly linear speedup factor of on 16 cores.

The other core routine computes the so-called *base monomial set* (Chapter 4), which is the vertex set of the above hypergraph. Therefore, the base monomial set directs the process of extracting partial syntactic factorizations from the input polynomial f. Different choices of base monomial sets produce different factorizations, and thus affect the evaluation cost. For the base monomial set, we choose the minimal elements of the set of all pairs of gcds of monomials in f, where the partial order is given by the divisibility relation on monomials.

In Chapter 9, we discuss the general problem of computing the minimal elements of an arbitrary partially ordered set. We propose a cache-friendly and parallel algorithm for this task; we also report on its application to transversal hypergraph generation. By cache-friendly, we mean that the serial counterpart of our parallel algorithm (obtained by ignoring the `spawn` and `sync` constructs in Algorithm 28 p. 86) has a better cache complexity than the straightforward method (Algorithms 27 p. 86) for the same task. More precisely, for an ideal cache of Z words, with cache lines of size L, on an input of n elements, the number of cache misses of the serialized version of our algorithm is $O(n^2/(ZL) + n/L)$ whereas the cache complexity of the straightforward method is $O(n^2/L)$.

For sufficiently large input hypergraphs, our `Cilk++` code reaches a linear speedup

factor on 32 cores. This latter work is reported in a paper [43] co-authored with Charles E. Leiserson, Marc Moreno Maza and Yuzhen Xie.

1.1.3 Evaluation Schedule and Experimental Results

Once a syntactic decomposition of the input polynomial has been obtained, we apply to it a widely used common subexpression elimination (CSE) technique, which turns a syntactic decomposition to a DAG and reduces the evaluation cost. Targeting multicore computer architectures, our next objective is to decompose the DAG constructed from the syntactic decomposition into p sub-DAGs for a specified parameter p, the number of available processors. These sub-DAGs should be independent to each other in the sense that the evaluation of one does not depend on the evaluation of the others. In this manner, these sub-DAGs can be assigned to different processors independently. Ideally, we also expect the size of these sub-DAGs to be balanced such that the "span" of the intended parallel evaluation is minimal. Our method for generating these sub-DAGs (see Chapter 7) provides an efficient evaluation schedule of the input polynomial in terms of work, data locality and parallelism.

The experimental results reported in Chapter 8 illustrate the effectiveness of our approach compared with other available software tools. For $2 \leq n, m \leq 8$, we have applied our techniques to the resultant $R(a, b)$ defined before. For $(n, m) = (7, 6)$, our resulting schedule evaluates 10 times faster than the direct evaluation of its straight-line program (SLP) representation (both conducted sequentially). For this case $(n, m) = (7, 6)$, no code optimization software tools we have tried produces a satisfactory result.

1.2 Organization of the Thesis

Related work and background notions are reviewed in Chapter 2. After introducing the concept of a syntactic decomposition in Chapter 3, the hypergraph method is presented in Chapter 4. The following chapter is devoted to estimating the algebraic complexity of the proposed algorithm. The parallelization and implementation of the hypergraph method are reported in Chapter 6. In Chapter 7, we discuss our strategy for generating the parallel evaluation schedule of our SLPs. The experimentation is reported in Chapter 8. In Chapter 9, we discuss the parallel computation of the minimal elements of a partially ordered set. We conclude this thesis by some future work directions in Chapter 10.

Chapter 2

Background and Literature Review

Encodings for multivariate polynomials in computer programs are at the core of this study. Thus we start this chapter by reviewing popular representations of multivariate polynomials. Section 2.1 is dedicated to those representations which are called *canonical*, that is, those representations which establish a one-to-one map between the mathematical objects and their computer representations. Section 2.2 is dedicated to straight-line programs (SLPs) as a way of representing polynomials. In this case, the representation is no longer canonical as two different SLPs may encode the same polynomial.

From Section 2.3, we discuss techniques that make the evaluation of polynomials more efficient. This includes the well-known Horner scheme and its generalizations, the parallel evaluation of DAGs and other related works.

2.1 Canonical Representation of Multivariate Polynomials

In computer programs, polynomials can be represented in various ways. These different representations facilitate different operations on polynomials. For example, dense representations provide fast retrieval of coefficients; however this feature is to the cost of recording every possible coefficient of the polynomial, no matter it is zero or not. Sparse representations save on memory consumption but this makes the access to coefficients less easy than with dense representations. In 1984 David Stoutemyer in [56] discussed different canonical representation of polynomials.

In the sequel of this chapter, we consider a commutative ring $\mathbb{R}$ with unity. We

denote by $\mathbb{X} = \{x_1, \ldots, x_n\}$ a finite set of variables. We review hereafter some of the classical representations of multivariate polynomials in $\mathbb{R}[\mathbb{X}]$.

2.1.1 Recursive Representation

If $n = 1$ we can use any possible univariate representations such as the following ones.

- For the polynomial $f = a_n x^n + \cdots + a_1 x + a_0$ one can use an array A of size $n+1$ indexed from 0 to n, such that the slot $A[i]$ contains the coefficient a_i, for all $i = 0 \cdots n$. Such representation is called *univariate dense*.
- For the polynomial $f = a_n x^n + \cdots + a_1 x + a_0$ one can use a sorted list of all pairs $[i, a_i]$ such that $a_i \neq 0$ holds. Such representation is called *univariate sparse*.

When $n > 1$, we can view $\mathbb{R}[\mathbb{X}]$ as the univariate polynomial ring $\mathbb{R}[x_1, \ldots, x_{n-1}][x_n]$ and proceed recursively. This implies to choose an ordering on the variables.

2.1.2 Sparse Distributed Representation

Each polynomial can be viewed as a linear combination of monomials (with coefficients in $\mathbb{R}$). Then the polynomial

$$p \;=\; a_1 m_1 + \cdots + a_p m_p$$

where the m_i are pairwise different monomials and the a_i are nonzero coefficients, can be represented as an aggregate of terms $[a_i, m_i]$. To have a fast equality-test, the aggregate of terms must be *linear* and sorted. In other words, there should be a *first* term, a *second* term, and so on. Thus this aggregate should better be a *list* or an *array* and the monomials should be totally ordered. Two types of monomial orderings are frequently used.

- The lexicographical ordering. With $\mathbb{X} = \{x > y > z\}$ we have

$$1 < z < \cdots < z^n \cdots < y < yz < \cdots < yz^n \cdots < y^2 < y^2 z < \cdots < y^2 z^n \cdots \tag{2.1}$$

- The degree-lexicographical ordering. With $\mathbb{X} = \{x > y > z\}$ we have

$$1 < z < y < x < z^2 < zy < y^2 < zx < xy < x^2 < \cdots < \tag{2.2}$$

2.2 Straight-Line Programs

In this section, we give a formal definition of straight-line programs. Our definition is in the spirit of others such as the one of [45], but slightly more general and thus simpler. This mathematical definition allows us to consider different possible computer representations for SLPs. The one used in our `Cilk++` code is described in Section 2.2.2.

Let $\mathbb{R}$ be a commutative ring with unity. Let V be a finite set of symbols $\{v_1, \ldots, v_\tau\}$ and let S be a finite sequence $(s_1, \ldots, s_\ell)$ of *assignments* where each s_i is one of the following forms:

- $v \leftarrow r$,
- $w \leftarrow u + v$,
- $w \leftarrow u - v$,
- $w \leftarrow u \times v$,
- $w \leftarrow u/v$, if $\mathbb{R}$ is a field,

where $r \in \mathbb{R}$ and $u, v, w \in V$.

Definition 1. *Let i be in the range $1 \cdots \ell$. The assignment s_i is said to be* valid *if the following condition holds:*

1. *if $1 < i \leq \ell$ holds and if a symbol $v \in V$ appears on the right-hand side of the assignment $s_i \in S$ then there exists an assignment $s_j \in S$ with $1 \leq j < i$ such that v appears on the left-hand side of the assignment s_j,*
2. *if $1 = i \leq \ell$ then the assignment s_i is of the form $v \leftarrow r$, for some $r \in \mathbb{R}$ and some $v \in V$.*

The assignment s_i is said to be useful *in the sequence $(s_1, \ldots, s_\ell)$ if one of the following conditions holds:*

1. *if $1 \leq i < \ell$ holds and if $v \in V$ is the symbol on the left-hand side of the assignment s_i, then there exists an assignment $s_j \in S$ with $i < j \leq n$ such that v appears on the right-hand side of the assignment s_j,*
2. *if $1 < i = \ell$ holds then the assignment s_i is not of the form $v \leftarrow r$, for some $r \in \mathbb{R}$ and some $v \in V$.*

Definition 2. *We say that the pair $P = (V, S)$ is a* straight-line program *(SLP for short) over* $\mathbb{R}$ *if for all $i = 1 \cdots \ell$, the statement s_i is valid and useful in S.*

Let $P = (V, S)$ be an SLP over $\mathbb{R}$. The requirement that every statement of S is useful and valid in S permits to establish Proposition 1 which endows our SLP with a recursive structure, leading to Definition 4. Another advantage of our formalism is that it supports a natural extension to SLPs with multiple output values.

We observe that, if $\ell > 1$ holds, then $(V, (s_1, \ldots, s_{\ell-1}))$ is not necessarily an SLP over $\mathbb{R}$. Consider for instance $V = \{x, y\}$ and $S = (x \leftarrow 1, y \leftarrow 1, y \leftarrow x + y)$.

Suppose that $\ell > 1$ holds. Then s_ℓ is of the form $x \leftarrow y \,\mathsf{op}\, z$, for $x, y, z \in V$ and where op is one of the binary arithmetic operators. Let i (resp. j) be the largest index in the range $1 \cdots \ell - 1$ such that s_i (resp. s_j) is an assignment whose left-hand side is x (resp. y). If s_i (resp. s_j) is of the form $v \leftarrow r$, for $v \in V$ and $r \in R$ we define $S^+ = (s_i)$ (resp. $S^- = (s_j)$). Otherwise, let S^+ (resp. S^-) be the subsequence of $(s_1, \ldots, s_i)$ (resp, $(s_1, \ldots, s_j)$) obtained by removing from $(s_1, \ldots, s_i)$ (resp, $(s_1, \ldots, s_j)$) any assignments which are not useful in $(s_1, \ldots, s_i)$ (resp, $(s_1, \ldots, s_j)$). Define $P^+ = (V, S^+)$ and $P^- = (V, S^-)$. The following proposition is easy to check.

Proposition 1. *Either $i = \ell - 1$ or $j = \ell - 1$ holds. Moreover P^+ and P^- are SLPs over* $\mathbb{R}$.

Definition 3. *Let $P = (V, S)$ be an SLP over* $\mathbb{R}$*. The integer ℓ is called the* length *of P and is denoted by* $\mathsf{length}(P)$*. The number of statements in P which are of the form $w \leftarrow u \,\mathsf{op}\, v$, with $u, v, w \in V$ and $\mathsf{op} \in \{+, -, \times, /\}$, is called the* work *and is denoted by* $\mathsf{work}(P)$*. The symbol of the left-hand side of s_ℓ is called the* output *of P and is denoted by* $\mathsf{output}(P)$*. The set of the elements $r \in \mathbb{R}$ that appear on the right-hand side of a rule of the form $v \leftarrow r$, for $v \in V$, is denoted by* $\mathsf{input}(P)$.

Definition 4. *Let $P = (V, S)$ be an SLP over* $\mathbb{R}$*. We define inductively the* value *of P, denoted by* $\mathsf{value}(P)$*, as follows:*

- *if $\mathsf{length}(P) = 1$, then there exists $(v, r) \in V \times \mathbb{R}$ such that $s_1 = v \leftarrow r$ and we define $\mathsf{value}(P) = r$,*
- *if $\mathsf{length}(P) > 1$, then s_ℓ is of the form $x \leftarrow y \,\mathsf{op}\, z$, for $x, y, z \in V$ and where op is one of the binary arithmetic operators. In this case we define*

$$\mathsf{value}(P) = \mathsf{value}(P^+) \,\mathsf{op}\, \mathsf{value}(P^-).$$

Observe that $\mathsf{value}(P)$ is an element of $\mathbb{R}$.

The purpose of Definition 5 is to obtain the following natural notion of equivalence between SLPs: two SLPs are equivalent if they have the same value, that is, essentially if they compute the same thing. This is formalized below.

Definition 5. *Let* $P_1 = (V_1, S_1)$ *and* $P_2 = (V_2, S_2)$ *be two SLPs over* $\mathbb{R}$. *We say that* P_1 *and* P_2 *are* equivalent *if we have* $\mathsf{value}(P_1) = \mathsf{value}(P_2)$.

To represent a polynomial of $\mathbb{K}[\mathbb{X}]$ using the previous concept of straight-line program in a natural manner, we can set $\mathbb{R}$ to $\mathbb{K}[\mathbb{X}]$ and impose that for each rule of the form $v \leftarrow r$ (with $r \in \mathbb{R}$) the right-hand side element r is a *literal*, that is, $r \in \mathbb{K} \cup \mathbb{X}$. Then the value of the SLP equals that of the represented polynomial.

2.2.1 Related Work on Straight-Line Program Representation

SLP representations can help reveal the structure of a polynomial expression. Consider the following multivariate polynomial proposed in [49]:

$$f = (x_1^2 + x_1 + 1)(x_2^2 + x_2 + 1) \cdots (x_n^2 + x_n + 1).$$

We can represent it by an SLP without expanding it. Note that the number of terms in this polynomial grows exponentially with n. Thus the size of a canonical representations of f grows exponentially with n while the length of an SLP representation grows linearly with n.

The advantages of SLP representations over classical representations have drawn many research interests. Here are some applications which have been studied in the literature:

- interpolation of polynomials given by this representation [25],
- solving zero-dimensional polynomial systems [28].
- computation of multihomogeneous resultants using SLPs [36],
- factorization of polynomials given by SLPs [38].
- greatest common divisors (GCDs) of polynomials [38, 49],

Using SLPs for computing algebraic objects that could not be computed otherwise, due to size issues, has been used in practice in a few other works, for example, for

computing the characteristic polynomial of multivariate polynomial matrices in [45]. There are also available computer algebra software packages for manipulating polynomials represented by SLPs, like Dagwood [22]. In [49], Bill Naylor reports on an Axiom domain which supports various operations on polynomials given by SLPs.

2.2.2 Implementation of Straight-Line Programs

We describe here our data structure for representing SLPs. For now, we do not allow subtraction or division with our SLPs. We represent an SLP by a flat array of lines, where each line is encoded by four integer fields, namely:

- `id`: this field records the index of the line in the array representing the whole SLP.
- `type`: the type of a line, which could be: variable in $\mathbb{X}$, element in $\mathbb{K}$, addition operation or multiplication operation.
- `operand1`: if the line is of type addition or multiplication, then this field records the `id` of its first operand, that is, the line whose computed value is `operand1`. If the line is an element in $\mathbb{K}$, than its value is recorded in this field. For line of type variable, the index of the variable in the variable list $\mathbb{X}$ is recorded in this field.
- `operand2`: If the line is of type addition or multiplication, then this field records the `id` of its second operand. For lines of type number or variable, this field is set to `NULL`.

Example 3. *Considering the following SLP over Z, the ring of integers.*

$$\begin{array}{rcl} x & \leftarrow & 1 \\ y & \leftarrow & 5 \\ u & \leftarrow & x + y \\ v & \leftarrow & x + y \\ w & \leftarrow & u \times v \end{array}$$

Its representation in our implementation is as follows:

In this encoding, each symbol (x, y, u, v or w) appearing at the left hand side of an assignment is now assigned to an `id`. At the implementation level, we define a macro for each type of a line. For example: number as 1, variable as 2, addition as

id	type	operand1	operand2
1	num	1	-
2	num	5	-
3	+	1	2
4	+	1	2
5	×	3	4

3 and multiplication as 4. If the id's are indexed from 1, then the above SLP can be encoded by the following array:

1	1	1	0	2	1	5	0	3	3	1	2	4	3	1	2	5	4	3	4

2.3 Optimizing the Evaluation of Polynomials

There are a number of approaches to minimize the evaluation cost of a polynomial expression. We review here related work on this topic, namely, common subexpression elimination techniques, the Horner scheme and the algorithms based on combinatorial structures. The algorithm presented in Chapter 4 presented in this thesis belongs in this latter category.

2.3.1 Common Subexpression Elimination

Conventional common subexpression elimination (CSE) techniques are well-known in the compiler design community These techniques search instances of identical subexpressions among an expression, so as to avoid duplicating the same subexpressions. For example, given two expressions,

$$\begin{aligned} f &= a + b + c + d \\ g &= a + b + e \end{aligned}$$

the recognition of the common subexpression $a + b$ would save one addition computation. However, by a straight left-to-right scan on the two statements

$$\begin{aligned} f &= a + b + c + d + e \\ g &= a + c + e \end{aligned}$$

no common subexpressions may be detected. In [11], Melvin A. Breuer discusses this problem and presents methods for finding common subexpressions to increase the efficiency of object code. However, as general-purpose applications, CSE techniques

are not suited for optimizing large polynomial expressions. In particular, they do not take full advantage of the algebraic properties of the expression to be optimized, when this expression is a polynomial or a matrix. However, they can always serve as a post-processing technique.

In the following we describe our implementation of CSE. By hashing expression lines, we expose multiple copies of the same expression. Recall that each line in an SLP is encoded by four integers (Subsection 2.2.2). The hash function used for an expression line is given below.[1]

```
unsigned hashFunction(unsigned *line) {
  int value;
  int list_length;
  switch(line[1]) {
  case num:
  case var:
    list_length = 2;  // [type, operand1]
    for(int i = 1; i < 3; ++i){
      value = (1000003 * value) ^ (line[i]);
      value = value ^ list_length;
    }
    break;
  case add:
  case mul:
    list_length = 3;  // [type, operand1, operand2]
    int op1 = (line[2] > line[3]) ? Line[2] : Line[3];
    int op2 = (line[2] > line[3]) ? Line[3] : Line[2];
    value = (1000003 * value) ^ (line[1]);
    value = value ^ list_length;
    value = (1000003 * value) ^ op1;
    value = value ^ list_length;
    value = (1000003 * value) ^ op2;
    value = value ^ list_length;
    break;
  default:
    printf("WRONG! line operation is %d \n", line[1]);
```

[1]The skeleton of this hash function can be found at http://effbot.org/zone/python-hash.htm.

```
  }
  if(value == -1)
    value = -2;
  return (unsigned)(value) % HashModuli;
}
```

We note that the commutativity of the addition and multiplication operations is supported by the above hash function. Indeed, this property is taken care of by the second and third lines after `case mul:`. However, the associative law is not supported. With this hash function, the algorithm for eliminating common subexpressions is straightforward and runs in linear time w.r.t. length of the input SLP. In Algorithm 1, L is an auxiliary array of length n, in which each entry $L[i]$ stores the id of S_i in the output SLP S'.

Input : an SLP S of length n
Output : an SLP S' has the same value as S but some common subexpressions are eliminated

```
1  S' ← ∅, n' ← 0;
2  L ← ∅, H ← empty hash table;
3  for i ← 1 to n do
4      set type of S'_{n'} same as S_i;
5      switch type of S_i do
6          case element in 𝕂
7              set the value of S'_{n'} same as S_i;
8          case variable in X
9              set the variable index of S'_{n'} same as S_i;
10         case addition or multiplication
11             let u be the id of the first operand of S_i;
12             let v be the id of the second operand of S_i;
13             set the id of the first operand of S'_{n'} as L_u;
14             set the id of the second operand of S'_{n'} as L_v;
15     if there exists S_j ∈ H computing the same expression as S'_{n'} then
16         L_i ← j;
17     else
18         add S'_{n'} to H;
19         L_i ← n';
20         n' ← n' + 1;
21 return S';
```

Algorithm 1: CommonSubexpressionElimination

2.3.2 Horner Scheme

One economic and popular approach to reduce the size of polynomial expression and facilitate their evaluation is the use of Horner's rule [34] named after William Horner. It transforms a polynomial into a sequence of nested additions and multiplications. For example, a polynomial in variable x

$$f = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

can be written in Horner's form as

$$f = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x))).$$

To evaluate f by this transformation, it requires n additions and n multiplications. Alexander Ostrowski [50] discussed in 1954 the optimality of Horner's rule in the evaluation of univariate polynomials. He conjectured that n multiplications or divisions are always required for the evaluation of a general univariate polynomial of degree n regardless of the number of additions or subtractions used. It is also conjectured by him that even if one has multiplications and divisions at free disposal, n additions or subtractions are necessary for the same evaluation problem. These conjectures can be proved by the method of E.G.Belaga [3] and T.S. Motzkin [48] and of Victor Pan [51]. Horner's rule matches these lower bounds and thus provides the smallest overall number of arithmetic operations for the evaluation of general univariate polynomials. Note that the transformation of Horner's rule contains a series of multiplications and additions that depend on the previous instruction. This makes it hard to be executed in parallel. Estrin's scheme is one method that attempts to introduce parallelism in this kind of scheme while increasing a bit the overall cost.

The multivariate Horner's rule first chooses a variable and apply Horner's rule regarding all the other variables as constants, then it selects one of the remaining variables and apply Horner's rule using this variable and so on. There are many different schemes [13, 14, 52, 53] to apply the multivariate Horner's rule. 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 variables. Unfortunately, there are $n!$ possible orderings for n variables.

Example 4 (Multivariate Horner Scheme)**.** *This example illustrates how the variable*

ordering affects the result of multivariate Horner's rule.

$$\begin{array}{rcll} f(a,b,c) & = & ab+bc+c & \\ (b>c>a) & = & b(a+c)+c & 1 \textit{ multiplication } +2 \textit{ additions} \\ (c>b>a) & = & ab+c(b+1) & 2 \textit{ multiplication } +2 \textit{ additions} \end{array}$$

2.3.3 Algorithms Based on Combinatorial Structures

It is a natural idea to consider algebraic factorization as a tool for minimizing the size of a polynomial expression. However, this fails for a variety of reasons discussed in Section 3. A work-around is to make use of combinatorial structures that reveal some patterns in the input polynomial expression. In [35], Anup Hosangadi, Farzan Fallah, and Ryan Kastner apply this principle. More precisely they generalize the algebraic techniques which have been successfully applied to reduce the number of literals in a set of Boolean expressions [7, 8, 9] to the optimization of polynomial expressions. Their method works in a way that extracts so-called kernels and co-kernels from a polynomial where the product of a kernel and the corresponding co-kernel equals part of the input polynomial. In this sense, kernels and co-kernels serve a role similar to our syntactic factorization (see Definition 7 p. 25).

The method presented in this thesis also fits in this category of algorithms based on combinatorial structures. In comparison to the work of [35], our method is more general. Indeed, our based monomial set (see Section 4.1 p. 29) which, in some sense, plays the role of their literals, is an input parameter of our method. This provides more flexibility. More importantly, our method runs in polynomial time w.r.t. the degree, number of variables and number of terms of the input polynomial, whereas the algorithm of [35] has a worst case which runs in singly exponential time w.r.t. the same quantities. This singly exponential behavior comes from a subroutine for common subexpression elimination (CSE). Our method does not need CSE at all. However, CSE can be used to post-process the output produced by our method. In our implementation, we do so and rely for that on a CSE procedure (Algorithm 1) which runs in linear time w.r.t. the lines of the input SLP.

2.4 Parallel Evaluation of DAGs

Several theoretical works have been done on the parallel evaluation of SLPs or algebraic circuits. In [10], Richard P. Brent presents a constructive proof on the parallelization of the parse tree of an expression. He has been shown that arithmetic

expression of size n can be rewritten in a straight line node of depth $O(\log n)$ under the assumption that all atoms (variables) in the arithmetic expression are distinct. By generalizing and parallelizing the sequential algorithms in [57], Gary L. Miller, Vijaya Ramachandran and Erich Kaltofen propose an algorithm to compress a DAG in a semi-ring in [46]. Their algorithm works on an arithmetic circuit satisfying the following three conditions or constraints: the first one is that there are no edges from multiplication nodes to multiplication nodes, the second one is that addition nodes can have any number of children [2] ,and the last one is that all edges are associated with certain weights. In the following paragraphs, we look at their main ideas in more detail.

The DAG evaluation is obtained by repeated applications of a procedure called *Phase*. Each phase consists of three steps, MM, $Eval_+$ and *shunt*. Step MM compresses addition chains (chains containing only addition nodes) by performing a matrix multiplication and matrix addition on the *connection matrix* of the circuit. Figure 2.1 and Figure 2.2 illustrate an SLP before and after applying procedure MM. Notice that the *plus-plus* edge, node 4 to node 5, has been compressed. Without specific mention, the edges in our examples have weight 1.

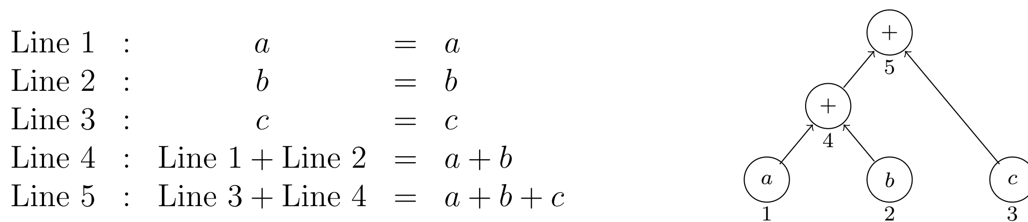


Figure 2.1: Input DAG of MM

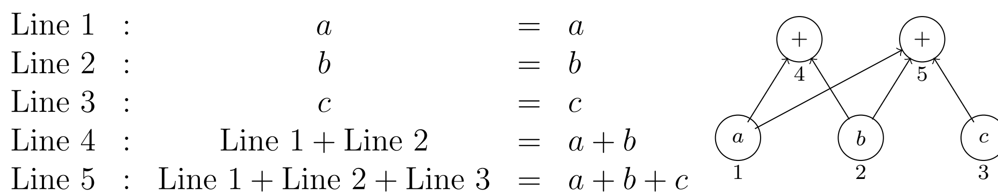


Figure 2.2: DAG after applying MM

Procedure $Eval_+$ evaluates addition nodes all of whose children are leaves and the evaluated nodes are then marked as leaves.

[2] A node u is a child node of the parent node v if there exists an edge from u to v.

The procedure *shunt* works on multiplication nodes in two steps. The first step works analogous to $Eval_+$ which evaluates multiplication nodes both of whose children are leaves and then regards them as leaves. The second step does partial evaluation of multiplication nodes dealing with the case where only one of its two children is a leaf.

For example for the DAG in Figure 2.3, node 6 (resp. node 7) has only one leaf child c (resp. d) and the other child node 5 is not a leaf. Then node 6 (resp. node 7) is partially evaluated in the Figure 2.4.

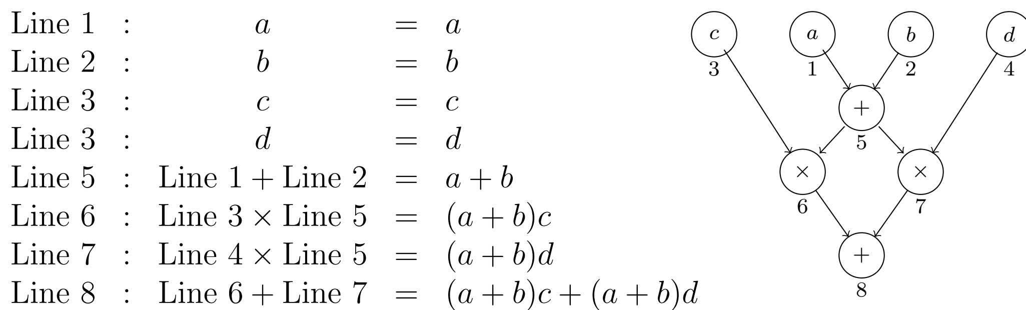


Figure 2.3: Input DAG of *shunt*

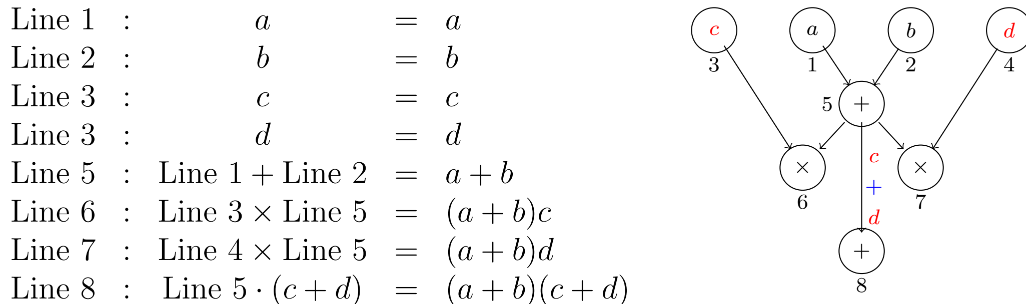


Figure 2.4: DAG after applying *shunt*

As pointed out by Gary L. Miller, Vijaya Ramachandran and Erich Kaltofen in [46], these procedures have strong analog to the procedure *Rake* and *Compress* for compressing expression trees in [47], where *Rake* is the operation to remove all leaves of a tree and *Compress* compresses chains (like MM in [46] compresses addition chains). The parallel complexity of [46] is analyzed by showing that each application of the procedure *phase* reduces the *height* of an arithmetic circuit by at least a half. Nathalie Revol and Jean-Louis Roch later extend this work to a larger variety of

algebraic structures including lattices in [54]. The classical PRAM model is used for analyzing the parallel algorithms mentioned in this subsection. In the rest of this thesis, we rely instead on the *fork-join multithreaded parallelism model* [24] which is well adapted to implementation on multicore architectures.

2.5 Other Related Work

Rob Corless, David Jeffrey and Michael Monagan propose a method to solve fluid-flow problems by using perturbation expansions, with special care of intermediate expression swell in [17]. They develop tools that represents and manipulates large expressions efficiently in Maple.

Chapter 3

Syntactic Decomposition

We introduce in this chapter the concepts used in our algorithm computing a "evaluation-efficient" representation of a given 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 $\mathsf{monoms}(f)$, thus f writes

$$f = \sum_{m \in \mathsf{monoms}(f)} c_m \, m,$$

where, for all $m \in \mathsf{monoms}(f)$, $c_m \in \mathbb{K}$ is the coefficient of f w.r.t. m. The set

$$\mathsf{terms}(f) = \{c_m \, m \mid m \in \mathsf{monoms}(f)\}$$

is the set of the terms of f. We use $\#\mathsf{terms}(f)$ to denote the number of terms in f.

One observation is that factoring a polynomial often reduces its evaluation cost. For example, directly evaluating

$$f = ac + ad + bc + bd$$

would take 4 multiplications and 3 additions. If we factor f as

$$f = (a + b)(c + d),$$

we reduce the evaluation cost to 1 multiplication and 2 additions. However, factoring a polynomial has the following limitations for our purpose of evaluating large polynomials. First of all, a given large polynomial is often irreducible which is the case for generic resultants. Secondly, factoring a polynomial does not always reduce

its evaluation cost. In the following example, the factored form of f requires more operations to evaluate f than its expanded form:

$$f = x^n - 1 = (x-1)(x^n + x^{n-1} + \cdots + 1).$$

We introduce the notion of a *syntactic factorization* which is guaranteed to never increase the evaluation cost.

Definition 6 (Syntactic operations)**.** *Let $g, h \in \mathbb{K}[X]$. We say that gh is a* syntactic product*, and we write $g \odot h$, whenever*

$$\#\mathsf{terms}(g\,h) = \#\mathsf{terms}(g) \cdot \#\mathsf{terms}(h) \tag{3.1}$$

holds, that is, if no combinations 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*

$$\#\mathsf{terms}(g+h) = \#\mathsf{terms}(g) + \#\mathsf{terms}(h) \tag{3.2}$$

(resp. $\#\mathsf{terms}(g-h) = \#\mathsf{terms}(g) + \#\mathsf{terms}(h)$).

Remark 1. *Note that if a product $f = g\,h$ is not a syntactic product, i.e. there monomials combine when multiplying g and h, then the number of terms in f is less than the product of the number of terms in g and h. That means Equation 3.1 does not hold for non-syntactic product. Similarly, Equation 3.2 does not hold for non-syntactic sum.*

Example 5. *Assume $\mathbb{K}$ is $\mathbb{Q}$ or $\mathbb{C}$. Then, $(a+b)(a+b)$ is not a syntactic product, neither $(a+b)(a-b)$, whereas $(a+b)(c+d)$ is one.*

Definition 7 (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 $\mathsf{monoms}(g) \subseteq \mathcal{M}$ both hold. In this case, we call g the* base *and h the* cofactor *of this syntactic factorization.*

Definition 8 (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 2 gives an obvious upper bound for $\mathsf{cost}(f)$.

Proposition 2. *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:*

$$\frac{\min(t_g, t_h)}{2} \left(1 + \mathsf{cost}(g) + \mathsf{cost}(h)\right) \leq t_f(d_f + 1) - 1. \tag{3.3}$$

Proof. The inequality $\mathsf{cost}(f) \leq t_f(d_f + 1) - 1$ follows from the fact that evaluating each of the t_f terms requires at most d_f multiplications. It is easy to show

$$t_g(d_g + 1) + t_h(d_h + 1) \leq t_f(d_f + 1), \tag{3.4}$$

with the equalities $d_f = d_g + d_h$, $t_f = t_g t_h$ and the condition $t_f > 1$. Then we have

$$\begin{aligned} \frac{t_f(d_f + 1) - 1}{1 + \mathsf{cost}(g) + \mathsf{cost}(h)} &\geq \frac{t_f(d_f + 1) - 1}{t_g(d_g + 1) + t_h(d_h + 1) - 1} \\ &\overset{(3.4)}{\geq} \frac{t_f(d_f + 1)}{t_g(d_g + 1) + t_h(d_h + 1)} \\ &\geq \frac{t_f(d_f + 1)}{t_g(d_f + 1) + t_h(d_f + 1)} \\ &= \frac{t_g t_h}{t_g + t_h} \geq \frac{\min(t_g, t_h)}{2}, \end{aligned}$$

which implies Relation (3.3). □

Proposition 2 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. Another potential speedup factor comes from the fact that $d_g + d_h = d_f$ holds.

Definition 9 (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 in Chapter 4. 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 6 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:

$$XZ = M,\ XT = P,\ YZ = N \quad \text{and} \quad YT = Q.$$

This implies that $M/P = N/Q$ holds. Then, one can check that

$$\left(g,\, g',\, \frac{M}{g}, \frac{N}{g'}\right)$$

is a solution for (X, Y, Z, T), where $g = \gcd(M, P)$ and $g' = \gcd(N, Q)$.

Consider another example: with $t_f = 6$ and $t_g = 3$ and $t_h = 2$, let

$$f = M + N + P + Q + S + T, g = X + Y + Z \text{ and } h = U + V.$$

Up to renaming the terms of f, the following system must have a solution:

$$XU = P,\ X\ V = S,\ \ YU = N,\ YV = R,\ \ ZU = M,\ ZV = Q.$$

This implies that $SM = QP$ and $SN = PR$ must hold and one can check that

$$\left(g, \frac{Rg}{S}, \frac{Qg}{S}, \frac{P}{g}, \frac{S}{g}\right)$$

is a solution for (X, Y, Z, U, V, V) where $g = \gcd(P, S)$.

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.

Chapter 4

The Hypergraph Method

In this chapter, we introduce our approach for computing a syntactic decomposition of a given polynomial. One should bear in mind that this method was designed with the objective of hadling very large input. This motivates the design and implementation of our algorithms. By means of a *hypergraph*, our main routine, presented in Section 4.1, extracts syntactic factorizations from the input polynomial. In section 4.2, we discuss the choice and calculation of an important quantity in the computation of a syntactic decomposition, the *base monomial set*. Finally, in Section 4.3, we state the top level algorithm to produce a syntactic decomposition.

4.1 Partial Syntactic Factorization

Based on the concluding remarks of Chapter 3 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 $\mathsf{terms}(p) \subseteq \mathsf{terms}(f)$ holds, 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 $\mathsf{ParSynFactorization}(f, \mathcal{M})$ (Algorithm 3 p. 34) states this strategy formally: the key idea is to consider a hypergraph $\mathsf{HG}(f, \mathcal{M})$ which allows us to detect "candidate syntactic factorizations".

Definition 10 (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. $\forall i = 1 \cdots e$, $\mathsf{monoms}(g_i) \subseteq \mathcal{M}$,

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 that this partial syntactic factorization is trivial *if every* $g_i \odot h_i$ *is a trivial syntactic factorization.*

Proposition 3. *If 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.* $\mathcal{M}$*, then* r *does not admit any nontrivial partial syntactic factorization w.r.t.* $\mathcal{M}$*.*

This proposition can be proved by a simple contradiction to the definition of a partial syntactic factorization.

Proposition 4. *Suppose* $\mathcal{M}$ *is a monomial set satisfying the following condition,*

$$\forall m_i, m_j \in \mathcal{M}, i \neq j \Rightarrow m_i \nmid m_j \tag{4.1}$$

i.e., no monomial divides any other monomials in $\mathcal{M}$*. Then a polynomial* f *satisfying* $\mathsf{monoms}(f) \subseteq \mathcal{M}$ *does not admit any nontrivial partial syntactic factorization w.r.t.* $\mathcal{M}$*.*

Proof. If f admits a nontrivial partial syntactic factorization w.r.t $\mathcal{M}$,

$$f = (g_1 \odot h_1) \oplus (g_2 \odot h_2) \oplus \cdots \oplus (g_e \odot h_e) \oplus r, \tag{4.2}$$

then for each product $g_i\, h_i$, $i = 1, 2, \ldots, e$, we have

$$\mathsf{monoms}(g_i\, h_i) \subseteq \mathsf{monoms}(f) \subset \mathcal{M}.$$

Then there exists a monomial of h_i such that after multiplying a monomial of g_i results a monomial of $\mathcal{M}$. This contradicts the assumption of set $\mathcal{M}$ as $\mathsf{monoms}(g_i) \subseteq \mathcal{M}$. $\square$

From now on and throughout this section, we assume that the base monomial set $\mathcal{M}$ satisfies Condition (4.1). A direct consequence of Proposition 4 is as follows. If (4.2) is a partial syntactic factorization of f w.r.t $\mathcal{M}$, then every g_i does not admit any nontrivial partial syntactic factorization itself w.r.t $\mathcal{M}$.

Definition 11 (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*

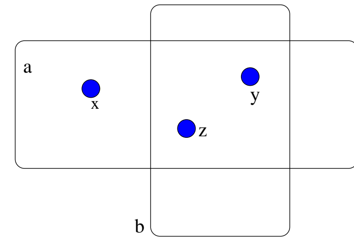
$$E_q := \{m \in \mathcal{M} \mid m\, q \in \mathsf{monoms}(f)\},$$

for an arbitrary monomial q. We say a hyperedge E_q is defined by monomial q and call q its defining monomial.

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*. Note that sum of all the isolated terms in f forms the polynomial r of Definition 10.

Example 6. *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*

$$\{(y + z, a + b), (x, a)\} \text{ and } aw.$$



For a given polynomial f and a given monomial set $\mathcal{M}$. the following algorithm constructs $\mathsf{HG}(f, \mathcal{M})$ straightforwardly.

```
Input  : a set of monomials M, the base monomial set 𝓜
Output : 𝓔, which is the set of hyperedges of hypergraph HG(f, 𝓜)
         where monoms(f) = M
1  Q ← ∅;
2  for m ∈ 𝓜 do
3      for m′ ∈ M do
4          if m | m′ then
5              q ← m′/m;
6              if q ∈ Q then
7                  E_q ← E_q ∪ {m};
8              else
9                  Q ← Q ∪ {q}; E_q ← {m};
10 𝓥 ← 𝓜; 𝓔 ← ∅;
11 for q ∈ Q do
12     𝓔 ← 𝓔 ∪ {E_q};
13 return 𝓔;
```

Algorithm 2: NaiveConstructHypergraph

The following proposition suggests how $\mathsf{HG}(f, \mathcal{M})$ can be used to compute a partial syntactic factorization of f w.r.t. $\mathcal{M}$.

Proposition 5. *Let $f, g, h \in \mathbb{K}[X]$ such that $f = g \odot h \oplus r$ and $\mathsf{monoms}(g) \subseteq \mathcal{M}$ both hold. Then, the intersection of all hyperedges E_q of $\mathsf{HG}(f, \mathcal{M})$, for $q \in \mathsf{monoms}(h)$, contains $\mathsf{monoms}(g)$.*

Proof. Denote by N the set of monomials such that

$$N = \{m\, q \mid m \in \mathsf{monoms}(g), q \in \mathsf{monoms}(h)\}.$$

It is clear from the definition of syntactic operation that,

$$|N| = |\mathsf{monoms}(g)| \cdot |\mathsf{monoms}(h)| \text{ and } N \subset \mathsf{monoms}(f).$$

Besides, we have $\mathsf{monoms}(g) \subseteq \mathcal{M}$. Thus for each $q \in \mathsf{monoms}(h)$, the hyperedge E_q contains all m for $m \in \mathsf{monoms}(g)$. This is followed by the definition of hypergraph $\mathsf{HG}(f, \mathcal{M})$ since $m \in \mathcal{M}$ and $m\, q \in \mathsf{monoms}(f)$ both hold. The conclusion follows. □

Before stating Algorithm ParSynFactorization, we make a simple observation.

Proposition 6. *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 $c = ab$. Suppose we are given two lists of monomials $G = \{G_1, G_2, \ldots, G_a\}$ and $H = \{H_1, H_2, \ldots, H_b\}$ such that the products $G_i H_j$ are all in $\mathsf{monoms}(f)$ 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 $f = g \odot h$, $\mathsf{monoms}(g) = G$ and $\mathsf{monoms}(h) = H$ 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 $\mathsf{monoms}(f)$. 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 $\mathbb{K}$. The conclusion follows. □

Remark 2. *Given a polynomial f and a monomial set $\mathcal{M}$, we are targeting at computing a syntactic factorization of f with respect to $\mathcal{M}$. Relying on the property of hypergraph $\mathsf{HG}(f, \mathcal{M})$ given by Proposition 5, we can extract two candidate monomial sets G and H from each intersection of hyperedges in $\mathsf{HG}(f, \mathcal{M})$. A point to make here is that with f, G and H at hand, one can check if there exists, and if it does, compute a syntactic factorization of f with respect to $\mathcal{M}$.*

The above number of bit operations for locating a monomial in the monomial set $\{F_1, F_2, \ldots, F_c\}$ only provides an upper bound. This estimate can surely be improved in various ways. For example, when the monomial set is sorted, we can locate a monomial in it by simply performing a binary search within $O(n \log c)$ bit operations. Another option is to introduce a suitable hash function: this will reduce the number of bit operations for locatong a monomial in a list to $O(n)$ bit operations.

The execution process of ParSynFactorization (Algorithm 3) is illustrated by Figure 4.1. It works in the following manner. Given a polynomial f and a monomial set

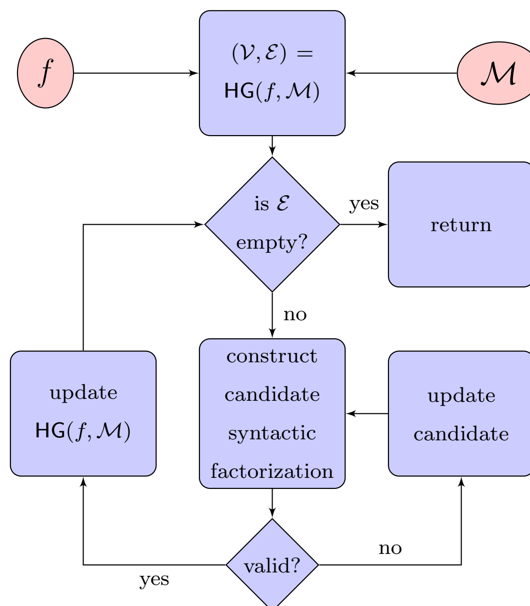


Figure 4.1: Computation process of partial syntactic factorization

$\mathcal{M}$, our objective is to compute a partial syntactic factorization of f with respect to $\mathcal{M}$. Recall that we expect this partial syntactic factorization to reduce the evaluating cost f as much as possible, while the construction remains practically cheap. To detect possible syntactic factorizations from f, we construct the hypergraph $\mathsf{HG}(f, \mathcal{M})$

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

```
1  T ← terms(f); F ← ∅;
2  r ← Σ_{t∈I} t where I = {t ∈ terms(f) | (∀m ∈ M) m ∤ t} ;
3  compute the hypergraph HG(f, M) = (V, E) by Algorithm 2;
4  while E is not empty do
5      if E contains only one edge E_q then
6          Q ← {q}; M ← E_q;
7      else
8          find q, q' such that E_q ∩ E_q' has the maximal cardinality;
9          M ← E_q ∩ E_q'; Q ← ∅;
10         if |M| < 1 then
11             find q such that E_q has the maximal cardinality;
12             M ← E_q; Q ← {q};
13         else
14             for E_q ∈ E do
15                 if M ⊆ E_q then
16                     Q ← Q ∪ {q} ;
17     while true do
18         N = {mq | m ∈ M, q ∈ Q};
19         if |N| = |M| · |Q| then
20             let p be the polynomial such that monoms(p) = N and
               terms(p) ⊆ T ;
21             if p = g ⊙ h with monoms(g) = M and monoms(h) = Q then
22                 compute g, h (Proposition 6);
23                 break;
24         else
25             randomly choose q ∈ Q, Q ← Q \ {q}, M ← ∩_{q∈Q} E_q;
26     for E_q ∈ E do
27         for m' ∈ N do
28             if q | m' then
29                 E_q ← E_q \ {m'/q} ;
30         if E_q = ∅ then
31             E ← E \ {E_q};
32     T ← T \ terms(p);
33     F ← F ∪ {g ⊙ h};
34 return F, r
```

Algorithm 3: ParSynFactorization

first. According to the property of this hypergraph given by Proposition 5, each intersection of hyperedges may lead to a syntactic factorization. We apply a greedy strategy here. The first attempt is to search for the largest intersection of any two hyperedges. If there exists a nonempty intersection M, we traverse each hyperedge to test if it is a superset of M and if it is, we add its defining monomial q to the set Q. When there does not exist nonempty intersection, we find the largest hyperedge and assign it to M while Q contains a single element, its defining monomial.

At this stage we have two lists of monomials M and Q at hand. They would potentially play the role of G and H in Proposition 6. However, before that happens, we should check whether they satisfy the assumption of Proposition 6 that there should be no combination of monomials when multiplying the monomials of M and Q (Line 18). This requirement is necessary to meet the definition of a syntactic factorization. It is possible that this requirement is not satisfied. For example, when $f = a^2 + 3ab + b^2$, we would get $M = Q = \{a, b\}$, which will lead to

$$|N| = 3 \neq 2 \times 2 = |M| \cdot |Q|.$$

If M and Q satisfies the assumption in Proposition 6, that is, $|N| = |M| \cdot |Q|$ holds, we find polynomial p such that $\mathsf{monoms}(p) = N$ and $\mathsf{terms}(p) \subseteq \mathsf{terms}(f)$. We can then follow the proof of Proposition 6 to find a syntactic factorization of p, satisfying

$$p = g \odot h, \mathsf{monoms}(g) = M \text{ and } \mathsf{monoms}(h) = Q.$$

It may also happen that a syntactic factorization satisfying these conditions does not exist, i.e., the system solving the coefficients of g and h as in the proof of Proposition 6 does not have solutions. The following example illustrates this case.

$$\mathcal{M} = \{a, b\}, Q = \{c, d\} \text{ and } p = ac + ad + bc + 2\,bd.$$

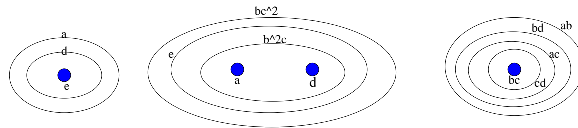
When the candidate M and Q does not meet the requirement of Proposition 6 or the system to solve the coefficients of g and h does not have a solution, we randomly remove an element from Q(Line 25) and retry to construct a syntactic factorization from M and current Q. We would also point out that the termination of the while loop in Line 17 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 6 always has a solution. Once a valid syntactic factorization has been built, 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 7 (Partial Syntactic Factorization)**.** *Consider*

$$f = 3ab^2c + 5abc^2 + 2ae + 6b^2cd + 10bc^2d + 4de + s \text{ and } \mathcal{M} = \{a, bc, e, d\}.$$

Following Algorithm 3, we first construct the hypergraph $\mathsf{HG}(f, \mathcal{M})$ *w.r.t. which the term s is isolated.*



It is easy to see that 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 therefore

$$\{mq \mid m \in M, q \in Q\} = \{ab^2c, abc^2, ae, b^2cd, bc^2d, de\}.$$

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 6, 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.

4.2 Base Monomial Set

One may notice that, until now, all our computation accepts two inputs, the given large polynomial and the so-called base monomial set $\mathcal{M}$. It is obvious that computations with different base monomial sets produce different partial syntactic factorizations, which will affect the number of operations to evaluate the input polynomial.

While it seems that this base monomial set is given by an oracle, there are actually many choices to construct an appropriate base monomial set for a given large polynomial. Let us first discuss what kind of properties this base monomial set should posses by reviewing how the base monomial set affects the computed partial syntactic factorization. Note that in Algorithm 3 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 $\mathsf{monoms}(g) \subseteq \mathcal{M}$. Therefore, to discover all the possible syntactic factorizations, the base monomial set should be chosen so as to contain all the monomials from which a syntactic factorization may be derived. We would also prefer the base monomial set to satisfy condition 4.1 such that the computation of partial syntactic factorization of all $g_i's$ can be avoided when we finalize the syntactic decomposition of the input polynomial (Proposition 4). To summarize, we would like the base monomial set satisfies the following two conditions:

- including all monomials from which a syntactic factorization may be derived;
- no monomials divides any other monomials of it (condition 4.1).

To this end, the most obvious choice is the set of all the variables X. However, to discover possible symmetries of a partial syntactic factorization, we may also consider

the set G of all non constant gcds of any two distinct monomials of f. However, $|G|$ could be quadratic in $\#\mathsf{terms}(f)$, which would be a bottleneck on large polynomials. Another disadvantage of set G is that it does not always satisfy condition 4.1. Our strategy is to choose for $\mathcal{M}$ as the set of the minimal elements of G for the divisibility relation. Algorithm 4 illustrates a straightforward way to compute this set. In theory, for base monomial computed by this algorithm, $|\mathcal{M}|$ fits in $|G| = O(t^2)$. In practice, $\mathcal{M}$ is much smaller than G (for large dense polynomials, $\mathcal{M} = X$ holds) and this choice is very effective. In [43], we formalize this problem as the computation of minimal elements in a partially ordered set and we extend discussion on this question in Chapter 6 and Chapter 9.

```
Input  : a set of monomials {M_1, ..., M_t}
Output : a base monomial set 𝓜
1  𝓜 ← ∅;
2  for i ← 1 to t do
3      for j ← i + 1 to t do
4          g ← Gcd(M_i, M_j); is_min ← true;
5          for m ∈ 𝓜 do
6              if m | g then
7                  is_min ← false;
8                  break;
9              else if g | m then
10                 𝓜 ← 𝓜 \ {m};
11         if is_min = true then
12             𝓜 ← 𝓜 ∪ {g};
13 return 𝓜;
```

Algorithm 4: NaiveBaseMonomialSet

4.3 Syntactic Decomposition

We report the top level algorithm to compute a syntactic decomposition of a given polynomial in this section. The algorithm keeps applying Algorithm 3 to cofactors of syntactic factorizations in its output. Serving as a subroutine, we assume that there is a procedure $\mathsf{ExpressionTree}(f)$ that outputs an expression tree of a given polynomial f. Algorithm 5, which we give for the only purpose of being precise, states the most straight forward way to implement $\mathsf{ExpressionTree}(f)$. In this algorithm, we denote

a tree by T, the left child of it by T_ℓ and the right child by T_r. We assume the following operations are valid on a tree. We can initialize a tree to contain a single node, which is usually a variable in X or a single element in $\mathbb{K}$. The children of a tree can be among variables, numbers and trees. The root of a tree is usually an operation $(+, \times)$ on its children unless the tree contains a single node. An obvious improvement of Algorithm 5 is to introduce the trick that reduces the nodes of the subtree representing x^n from $O(n)$ to $O(\log n)$, for example, x^8 can be represented by $((x^2)^2)^2$ rather than $x \cdot x \cdot \dots \cdot x$.

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

```
1  if ♯terms(f) = 1 say f = c · x_1^{d_1} x_2^{d_2} ··· x_k^{d_k} then
2      for i ← 1 to k do
3          T_i ← x_i;
4          for j ← 2 to d_i do
5              T_{i,ℓ} ← T_i, root(T_i) ← ×, T_{i,r} ← x_i;
6      root(T) ← ×, T_ℓ ← c, T_r ← T_1;
7      for i ← 2 to k do
8          T_ℓ ← T, root(T) ← ×, T_r ← T_i;
9  else
10     k ← s/2;
11     f_1 ← Σ_{i=1}^{k} t_i; f_2 ← Σ_{i=k+1}^{s} t_i;
12     T_1 ← ExpressionTree(f_1);
13     T_2 ← ExpressionTree(f_2);
14     root(T) ← +, T_ℓ ← T_1, T_r ← T_2;
```

Algorithm 5: ExpressionTree

Algorithm 6 formally states how to produce a syntactic decomposition of a given polynomial. Given a input polynomial f, we first compute the base monomial set $\mathcal{M}$ by Algorithm 4. If this set is empty, then no interesting syntactic decomposition could be derived from f, the algorithm terminates. Otherwise, we compute a partial syntactic factorization of f w.r.t. $\mathcal{M}$ by Algorithm 3, say

$$f = (g_1 \odot h_1) \oplus (g_2 \odot h_2) \oplus \cdots \oplus (g_e \odot h_e) \oplus r.$$

By Proposition 3, Proposition 4 and our choice of the base monomial set, neither $g_i's$ for $i = 1, \dots, e$ nor r admits any nontrivial partial syntactic factorization w.r.t. $\mathcal{M}$,

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

```
1  M ← base monomial set computed from f;
2  if M = ∅ then
3      return ExpressionTree(f);
4  else
5      F, r ← ParSynFactorization(f, M);
6      for i ← 1 to |F| do
7          (g_i, h_i) ← F_i;
8          T_i ← empty tree, root(T_i) ← ×;
9          T_{i,ℓ} ← ExpressionTree(g_i);
10         T_{i,r} ← SyntacticDecomposition(h_i);
11     root(T) ← +, T_ℓ ← ExpressionTree(r), T_r ← T_1;
12     for i ← 2 to |F| do
13         T_ℓ ← T, root(T) ← +, T_r ← T_i;
```

Algorithm 6: SyntacticDecomposition

whereas it is possible that one of $h_i's$ admits one. Like in Example 7,

$$h = 3b^2c + 5bc^2 + 2e$$

admits a nontrivial partial syntactical factorization,

$$h = bc \odot (3b + 5c) \oplus 2e.$$

Computing it will produce a syntactic decomposition of the input polynomial. 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. For those $h_i's$ which may derive syntactic decompositions themselves, we recursively call the same algorithm on them. The rest of the algorithm is just routine to combine those DAGs representing $g_i's$, $h_i's$ and r.

Chapter 5

Complexity Estimates

Given a polynomial f of t terms with total degree d in $\mathbb{K}[X]$, where $X = \{x_1, x_2, \ldots, x_n\}$, we analyze the running time for Algorithm 6 to compute a syntactic decomposition of f. We assume that the input polynomial is given in distributed representation and that each exponent in a monomial is encoded by a machine word. Thus 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. We estimate the time complexity of all algorithms although some of them may seem straight forward.

5.1 Monomial Set Construction

A base monomial set simply chosen as $\{x_1, x_2, \ldots, x_n\}$ can be easily obtained in $O(1)$ operations and $\mu = n$. Suppose a base monomial is computed by algorithm 4. There are $O(t^2)$ iterations of the two nested for loops in Line 2 to Line 12, during each of which at most one element is added to set $\mathcal{M}$. Therefore $\mu = O(t^2)$. The third nested for loop in Line 5 to Line 10 iterates μ times. Overall, algorithm 4 takes $O(t^2 \cdot \mu \cdot n) = O(t^4 n)$ bit operations to produce a base monomial set of size $O(t^2)$.

5.2 Hypergraph Construction

Assume that we have computed a base monomial set $\mathcal{M}$. Algorithm 2 constructs the hypergraph $\mathsf{HG}(f, \mathcal{M})$ by traversing $\mathcal{M}$ and all the monomials in f. In Line 2 to Line 9, there are $\mu \cdot t$ iterations during each of which at most one element is added to Q in $O(n)$ operations. This leads to the result that $|\mathcal{E}| = |Q| \in O(\mu t)$ which also gives

the number of operations in the last for loop in Line 11. Hence algorithm 2 constructs the hypergraph $\mathsf{HG}(f, \mathcal{M})$ of μ vertices and $O(\mu t)$ edges in $O(\mu t n)$ bit operations.

5.3 Computing Partial Syntactic Factorization

We proceed analyzing Algorithm 3. 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 $\mu t n$ 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 of any two hyperedges (Line 8).
- 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 11).
- 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 14 to Line 16).
- Line 8 to Line 16 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 18).
- by Proposition 6, 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 20 to Line 23).
- 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 18 to Line 23. 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 17 to Line 25).
- We update the hypergraph by removing the monomials in the constructed syntactic factorization. The two nested for loops in Line 26 to Line 31 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 $\mathsf{monoms}(f)$ is removed from the hypergraph $\mathsf{HG}(f, \mathcal{M})$. So the while loop from Line 4 to Line 33 would terminate in t iterations.

Overall, Algorithm 3 takes $O(\mu^5 t^4 n)$ bit operations and $O(\mu^3 t^3)$ operations in $\mathbb{K}$. In the case when terms of the input polynomial are sorted, the number of bit operations to compute or reject a candidate syntactic factorization is reduced to $O(|M||Q| \cdot \log(|M||Q|) \cdot n) = O(\mu^2 tn \log(\mu t))$ (Lines 20 to Line 23). As a consequence, the whole algorithm performs $O(\mu^3 t^3 n \log(\mu t))$ bit operations and $O(\mu^3 t^3)$ operations in $\mathbb{K}$.

5.4 Expression Tree Construction

One easily checks from Algorithm 5 that an expression tree can be computed from f (where f has t terms and total degree d) within in $O(ndt)$ bit operations.

Indeed, let $S(t, d, n)$ be the number of operations to produce an expression tree of f. Then it is easy to see that it satisfies the following recurrence relation

$$S(t, d, n) \leq \begin{cases} 2S(t/2, d, n) + O(1) & \text{if} \quad t > 1, \\ nd & \text{if} \quad t = 1, \end{cases}$$

Therefore, it can be computed in $O(ndt)$ operations.

5.5 Computing Syntactic Decomposition

In the sequel of this chapter, we analyze Algorithm 6. 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 9) and of the isolated polynomial r (Line 11) can be computed within $O(ndt)$ operations. Now, we shall establish an equation that rules the running time of Algorithm 6. Assume that $\mathcal{F}$ in Line 5 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 6 on all h_i's. Let $T_b(t, d, n)$ and $T_{\mathbb{K}}(t, d, n)$ be the number of bit operations and operations in $\mathbb{K}$ performed by Algorithm 6 respectively. We have the following recurrence

relation,

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

from which we derive that $T_b(t,d,n)$ is within $O(\mu^5 t^4 nd)$ 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 $O(t^{14} nd)$ 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 $\mathbb{K}$.

When terms of the input polynomial are sorted, we have

$$T_b(t,d,n) \in O(\mu^3 t^3 nd \log(\mu t)) \text{ and } T_{\mathbb{K}}(t,d,n) \in O(\mu^3 t^3 d).$$

Therefore, if the base monomial set is computed by Algorithm 4, then the number of bit operations and operations in $\mathbb{K}$ to compute a syntactic decomposition of a given polynomial is within $O(t^9 nd \log t)$ and $O(t^9 d)$. When base monomial set is chosen as the set of all the variables X, then these upper bounds become $t^3 n^4 d \log(tn)$ and $t^3 n^3 d$ respectively.

Theorem 1. *Given a polynomial of t terms with total degree d in $\mathbb{K}[X]$, where $X = \{x_1, x_2, \ldots, x_n\}$, then a syntactic decomposition of it can be computed within $O(t^9 nd \log t)$ bit operations and $O(t^9 d)$ operations in $\mathbb{K}$.*

Chapter 6

Implementation and Parallelization of the Hypergraph Method

As parallel hardware architectures (multi-cores, graphics processing units, etc.) and computer memory hierarchies (from processor registers to hard disks via successive cache memories) become more dominant in the modern computer, we focus our effort on improving both the parallelism and the cache-efficiency of our algorithms and their implementation. In this chapter, we parallelize the Algorithm SyntacticDecomposition (Algorithm 6 in p.40) and its subroutines without analyzing their parallelism and data locality. This analysis would essentially follow the same principle as the one conducted in Chapter 9 for the parallel computations of the minimal of a partially ordered set. Indeed, the main algorithms of the present chapter and Chapter 9 are *divide-and-conquer* algorithms where the *conquer* part fundamentally reduces to the parallel enumeration of the Cartesian product of two finite sets.

6.1 Data Structures

Let f be a multivariate polynomial in $X = \{x_1, x_2, \ldots, x_n\}$ over a field $\mathbb{K}$. To simplify the implementation and analysis, we assume that $\mathbb{K}$ is a finite field $\mathbb{Z}/p\mathbb{Z}$ where p is a machine prime number [1], and that partial degrees in any variable fits into a machine integer. In what follows, we list the major data structures used in computing the syntactic decomposition of f.

[1] This assumption is irrelevant to the theory but to the implementation, and can be easily relaxed once operations on big integers are provided (in software level, for example use the GNU Multiple Precision Arithmetic Library).

6.1.1 Monomials, Terms and Polynomials

A monomial is encoded by its exponent vector of length n and a term is encoded by an array of length $n+1$, where the first slot stores the coefficient, followed by the exponent vector of its supporting monomial. For example, monomial $ab^2c \in \mathbb{K}[a, b, c, d, e, s]$ is encoded as $[1, 2, 1, 0, 0, 0]$ and term $3ab^2c$ is encoded as $[3, 1, 2, 1, 0, 0, 0]$.

The following operations on monomials are supported:

Gcd: computes the gcd of two monomials by taking the smaller value of each entry in their exponent vectors;

Mul: computes the product of two monomials by adding two exponent vectors;

Same: checks if two monomials are the same;

Compare: checks the divisibility of two monomials, that is, Compare(m_1, m_2) returns 1 if m_1 divides m_2 (i.e. m_2 is a multiple of m_1), returns -1 if m_2 divides m_1, otherwise, returns 0 if m_1 and m_2 are incomparable.

A polynomial is represented by a list of terms (sparse distributed representation reviewed in Subsection 2.1.2). In the implementation, these terms are sorted with respect to the pure lexicographical order.

6.1.2 Hypergraphs

A hypergraph is defined by a vertex set and a set of hyperedges. The former is encoded as a list of monomials and a hyperedge is a collection of vertices defined by its defining monomial. Thus the fields of a *Hypergraph* class are

- a set of vertices, implemented as an array of monomials;
- a set of hyperedges, implemented as an array of pointers to the *Hyperedge* objects defined below.

A *Hyperedge* is a class containing two fields:

- the defining monomial, implemented as an exponent vector,
- a set of vertices, implemented as a bitvector object. Class *bitvector* supports the following standard operations:

  set: sets a bit,

unset: unsets a bit,

is_set: checks if a bit is set,

count: counts the number of set bits,

union: computes the union of two bitvectors,

intersect: computes the intersection of two bitvectors,

is_superset: tests the inclusion of two bitvectors.

Class *Hyperedge* supports the following operations:

Intersection: intersects two hyperedges by taking the intersection of two bitvectors.

Is_included: checks if a set of vertices is a subset of a hyperedge. This function accepts a bitvector, which encodes a set of vertices, and a *Hyperedge* object. It calls is_superset.

Note that the introduction of bitvectors allows efficient computation on hyperedges (Line 8, and Line 14 to Line 16 of Algorithm 3).

6.1.3 Syntactic Decomposition

The class *SynDecomposition* is designed to support the following partial syntactic factorization:

$$f = (g_1 \odot h_1) \oplus (g_2 \odot h_2) \oplus \cdots \oplus (g_e \odot h_e) \oplus r.$$

Recall that once a partial syntactic factorization are obtained, we recursively apply Algorithm 6 on all cofactors $h_i's$. Class *SynDecomposition* has the following fields:

cofactor: the cofactor of a syntactic factorization as a list of terms,

cof_size: the number of terms included in the cofactor,

base: the base of a syntactic factorization as a list of terms,

base_size: the number of terms included in the base,

isolated_terms: the isolated polynomial r as a list of terms,

iso_size: the number of terms included in the polynomial r,

syntactic_factors: a list of e pointers to *SynDecomposition* objects, where each object encodes a syntactic factorization $g_i \odot h_i$,

syn_size: the number of syntactic factorizations.

We note that *SynDecomposition* object does *not* form a binary tree as defined in the Definition 9. As illustrated by the following example, a *SynDecomposition* object will not be converted into an expression tree in this stage. The conversion to a binary tree is performed when the final evaluation SLPs are generated (Chapter 7). In Example 7, a syntactic decomposition of

$$f = 3ab^2c + 5abc^2 + 2ae + 6b^2cd + 10bc^2d + 4de + s$$

can be written as

$$(a + 2d) \odot (bc \odot (3b + 5c) \oplus 2e) \oplus s.$$

We illustrate the construction of a syntactic decomposition object by the this example. The input polynomial is encoded as

cofactor	:	$[3ab^2c, 5abc^2, 2ae, 6b^2cd, 10bc^2d, 4de, s]$
cof_size	:	7
all others	:	NULL

The base monomial set computed by Algorithm 4 is $\{a, bc, e, d\}$. It is encoded by a list of monomials. The hypergraph computed by Algorithm 2 contains the following hyperedges:

$$\begin{array}{rcccl} E_{b^2c} = E_{bc^2} = E_e & = & \{a, d\} & : & [1, 0, 0, 1] \\ E_{ab} = E_{ac} = E_{bd} = E_{cd} & = & \{bc\} & : & [0, 1, 0, 0] \end{array}$$

Once the hypergraph is constructed, the isolated term s is identified. By applying Algorithm 3, we extract a syntactic factorization $(a + 2d) \odot (3b^2c + 5bc^2 + 2e)$, which itself will be encoded as a *SynDecomposition* object (say poly1) and a pointer will be added to the *syntactic_factors* field of the root *SynDecomposition* object,

cofactor	:	$[3ab^2c, 5abc^2, 2ae, 6b^2cd, 10bc^2d, 4de, s]$
cof_size	:	7
base	:	NULL
base_size	:	0
isolated_terms	:	$[s]$
iso_size	:	1
syntactic_factors	:	[pointer to poly1]
syn_size	:	1

The field *syntactic_factors* is an array of pointers to *SynDecomposition* objects. This array now contains one pointer to the syntactic factorization $(a+2d)\odot(3b^2c+5bc^2+2e)$ (poly1). To complete the construction of the syntactic decomposition, we apply Algorithm 6 to poly1

cofactor	:	$[3b^2c, 5bc^2, 2e]$
cof_size	:	3
base	:	$[a, 2d]$
base_size	:	2
isolated_terms	:	$[2e]$
iso_size	:	1
syntactic_factors	:	[pointer to poly2]
syn_size	:	1

where *SynDecomposition* object poly2 is

cofactor	:	$[3b, 5c]$
cof_size	:	2
base	:	$[bc]$
base_size	:	1
all others	:	NULL

To sum up, the syntactic decomposition of a polynomial can be obtained recursively in the following manner:

$$\begin{cases} (\text{cofactor} + \sum isolated_terms)(\sum base) & \text{if syntactic_factors is NULL,} \\ (\sum syntactic_factors + \sum isolated_terms)(\sum base) & \text{otherwise.} \end{cases} \tag{6.1}$$

6.2 Parallelization

Before describing the parallelization in more details let us first specify the programming model. We adopt the multi-threaded programming model of Cilk [24]. In our pseudo-code, the keywords **spawn** and **sync** have the same semantics as the **cilk_spawn** and **cilk_sync** statement in the `Cilk++` programming language [16]. A **cilk_spawn** tells the compiler that the function may run in parallel with the caller and a **cilk_sync** statement indicates that the current function can resume its execution once all functions spawned in its body have completed.

Our algorithms are also cache-efficiency oriented. Most of our algorithms follow the cache-oblivious philosophy introduced in [23]. More precisely, and similarly to the matrix multiplication algorithm of [23], they proceed in a divide-and-conquer fashion such that when a sub-problem fits into the cache, then all subsequent computations can be performed with no further cache misses. However, we also use a threshold such that, when the size of the input is within this threshold, then a base case subroutine is called. In principle, this threshold can be set to the smallest meaningful value, say 1, and thus our algorithms is cache-oblivious. In a software implementation, this threshold should be large enough so as to reduce parallelization overheads and recursive call overheads. Meanwhile, this threshold should be small enough in order to guarantee that, in the base case, cache misses are limited to cold misses. In the implementation of the matrix multiplication algorithm of [23], available in the `Cilk++` distribution, a threshold is used for the same purpose.

6.2.1 Merging Two Sets of Minimal Elements

We will first introduce a key building block, MinMerge, of the upcoming parallelized algorithms. It is frequently used in most of our algorithms. Intuitively, it works on two input sets both of which possess some property. We use this procedure to merge these two sets and obtain a set satisfying the same property. It was originally reported in [43] to merge two sets of minimal elements on a partially ordered set. We generalize it to a more general case where the partial order is relaxed to an arbitrary binary relation denoted by $\preceq$ on a set $\mathcal{X}$. We use of notion of "minimal" as in [43] such that an element a is minimal w.r.t a subset of $\mathcal{X}$, A, whenever for all elements $a' \in A$, we have $a' \preceq a \Rightarrow a' = a$. We say a subset A of $\mathcal{X}$ is "clean" whenever all its elements are minimal w.r.t it. The procedure MinMerge works on two clean input sets, say B and C. It outputs two clean sets E and F, such that for each element $b \in B$,

- if for all elements $c \in C$, either $b \preceq c$ or b incomparable to c holds, than $b \in E$.
- if there exists an element $c \in C$, such that $c \preceq b$ then the information of b is merged to c by a function Merge and b will not be included in E.

Similar relation holds for F and C.

We assume that the subsets of $\mathcal{X}$ are implemented by a data-structure which supports the following operations for any subsets A, B of $\mathcal{X}$:

Split: if $|A| \geq 2$ then Split(A) returns a partition A^-, A^+ of A such that the potential work to be done on A^- and A^+ are balanced. Hereafter, without explicit mention, it means that $|A^-|$ and $|A^+|$ differ at most by 1.

Union: Union(A, B) accepts two disjoint sets A, B and returns C where $C = A \cup B$;

In addition, we assume that each subset A of $\mathcal{X}$, with $k = |A|$, is encoded in a *C/C++ fashion* by an array `A` of size $\ell \geq k$. Elements in A would also support the following operations when applicable:

Mark: An element in A can be marked at trivial cost.

Merge: When an element $b \preceq a$ is detected, there is a function Merge(a, b) that merge the information of a to b. This function is only needed for some applications. If not mentioned, it does nothing.

When we refer to the following two algorithms (Algorithm 7 and Algorithm 8) on certain element type, we would specify how Split, Union, Mark and Merge are performed.

```
    Input: two "clean" sets B, C
 1  if |B| = 0 or |C| = 0 then
 2      return (B, C);
 3  else
 4      for i ← 1 to |B| do
 5          for j ← 1 to |C| do
 6              if c_j is unmarked then
 7                  if c_j ⪯ b_i then
 8                      Merge(b_i, c_j), Mark b_i and break inner loop;
 9                  if b_i ⪯ c_j then
10                      Merge(c_j, b_i), Mark c_j;
11      B ← {unmarked elements in B};
12      C ← {unmarked elements in C};
13      return (B, C);
```

Algorithm 7: SerialMinMerge

The straight forward Algorithm 7 presents a trivial way to achieve the objective described above. However, it can not be parallelized while taking advantage of the sparsity in the output. In addition, unless the input data fits in cache, Algorithm 7 is not cache-efficient. We apply a divide-and-conquer strategy in the parallelized

Input: two "clean" sets B, C

```
1  if |B| ≤ MIN_MERGE_BASE and |C| ≤ MIN_MERGE_BASE then
2      return SerialMinMerge(B, C);
3  else if |B| > MIN_MERGE_BASE and |C| > MIN_MERGE_BASE then
4      (B⁻, B⁺) ← Split(B);
5      (C⁻, C⁺) ← Split(C);
6      (B⁻, C⁻) ← spawn ParallelMinMerge(B⁻, C⁻);
7      (B⁺, C⁺) ← spawn ParallelMinMerge(B⁺, C⁺);
8      sync;
9      (B⁻, C⁺) ← spawn ParallelMinMerge(B⁻, C⁺);
10     (B⁺, C⁻) ← spawn ParallelMinMerge(B⁺, C⁻);
11     sync;
12     return (Union(B⁻, B⁺), Union(C⁻, C⁺));
13 else if |B| > MIN_MERGE_BASE and |C| ≤ MIN_MERGE_BASE then
14     (B⁻, B⁺) ← Split(B);
15     (B⁻, C) ← ParallelMinMerge(B⁻, C);
16     (B⁺, C) ← ParallelMinMerge(B⁺, C);
17     return (Union(B⁻, B⁺), C);
18 else
19     (C⁻, C⁺) ← Split(C);
20     (B, C⁻) ← ParallelMinMerge(B, C⁻);
21     (B, C⁺) ← ParallelMinMerge(B, C⁺);
22     return (B, Union(C⁻, C⁺));
```

Algorithm 8: ParallelMinMerge

Algorithm 8. To this end, we discuss the following four cases depending on the values of $|B|$ and $|C|$ w.r.t. the threshold MIN_MERGE_BASE.

Figure 6.1: Illustration of Algorithm 8 ParallelMinMerge

Case 1: both $|B|$ and $|C|$ are no more than MIN_MERGE_BASE which ensures that the computation of Algorithm 8 on $|B|$ and $|C|$ can completely fit in cache. We simply call the operation SerialMinMerge of Algorithm 7 The minimal elements from B and C are stored separately in an ordered pair (the same order as in the input pair) to remember the provenance of each result. In Cases 2, 3 and 4,

this output specification helps clarifying the successive cross-comparisons when the input posets are divided into subsets.

Case 2: both $|B|$ and $|C|$ are greater than MIN_MERGE_BASE. We split B and C into balanced pairs of subsets B^-, B^+ and C^-, C^+ respectively. Then, we recursively merge these 4 subsets, as described in Lines 6–10 in Algorithm 8. Merging B^-, C^- and merging B^+, C^+ can be executed in parallel without data races. These two computations complete half of the cross-comparisons between B and C. Then, we perform the other half of the cross-comparisons between B and C by merging B^-, C^+ and merging B^+, C^- in parallel. At the end, we return the union of the subsets from B and the union of the subsets from C. The left graph in Figure 6.2.1 illustrates this case.

Case 3, 4: either $|B|$ or $|C|$ is greater than MIN_MERGE_BASE, but not both. Here, we split the larger set into two subsets and make the appropriate cross-comparisons via two recursive calls, see Lines 14–22 in Algorithm 8. The right graph in Figure 6.2.1 illustrates this case.

We will back on the computation of minimal elements at Chapter 9, where the parallelism of Algorithm 8 and cache complexity of its C elision will be analyzed.

6.2.2 Computation of Base Monomial Set

We report the parallel computation of the base monomial set in this subsection. Recall that the base monomial set is chosen as the minimal elements of the pairwise Gcd's of all monomials in the input polynomial where the partial order is defined as divisibility. Intuitively, given a set of t monomials, one can first compute the list G of pairwise gcds of this set and then pass G to existing algorithm to compute the minimal elements, for example, Algorithm 28 in Chapter 9. However, G can be too large to fit into memory because $|G| = t(t-1)/2$. It then becomes desirable to compute the minimal elements concurrently to the generation of the Gcd set itself, thus avoiding storing the entire Gcd set in memory. Applying this strategy allows computations that could not be conducted otherwise. Moreover, by introducing a divide-and-conquer strategy, the proposed parallel algorithm becomes more cache friendly than Algorithm 4. The top-level routine is Algorithm 9 which takes as input a set A of monomials. In practice one would first call this routine with $A = \mathsf{monoms}(f)$. Algorithm 9 integrates the computation of $\mathcal{G}$ and $\mathcal{M}$ (as defined above) into a "single pass" divide-and-conquer process.

We now describe our divide-and-conquer method for computing the base monomial set of A, that is, $\mathsf{Min}(\mathcal{G}_A)$, where $\mathcal{G}_A$ consists of all non-constant $\gcd(a_1, a_2)$ for $a_1, a_2 \in A$ and $a_1 \neq a_2$. The top-level function is ParallelBaseMonomial of Algorithm 9. If $|A|$ is within a threshold, namely MIN_BASE, the operation SerialInnerBaseMonomials(A) is called. Otherwise, we partition A as $A^- \cup A^+$ and observe that

$$\mathsf{Min}(\mathcal{G}_A) = \mathsf{Min}(\mathsf{Min}(\mathcal{G}_{A^-}) \cup \mathsf{Min}(\mathcal{G}_{A^+}) \cup \mathsf{Min}(\mathcal{G}_{A^-,A^+}))$$

holds where $\mathcal{G}_{A^-,A^+}$ consists of all non-constant $\gcd(x, y)$ for $(x, y) \in A^- \times A^+$. Following the above formula, our parallel algorithm proceed in the following manner.

- Top level Algorithm 9 creates two independent computational branches:
  - one for $\mathsf{Min}(\mathsf{Min}(\mathcal{G}_{A^-}) \cup \mathsf{Min}(\mathcal{G}_{A^+}))$ which is computed by the operation SelfBaseMonomials of Algorithm 10;
  - one for $\mathsf{Min}(\mathcal{G}_{A^-,A^+})$ which is computed by the operation CrossBaseMonomials of Algorithm 11.

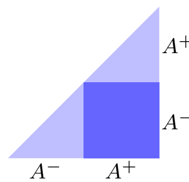


Figure 6.2: Illustration of Algorithm 9 ParallelBaseMonomials

- Algorithm 10 makes two recursive calls in parallel, then merges their results with Algorithm 8.

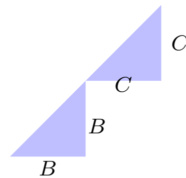


Figure 6.3: Illustration of Algorithm 10 SelfBaseMonomials

- In Algorithm 11 for CrossBaseMonomials, we uses a threshold, when the computation fits in cache, we do the computation serially by calling Algorithm 13. Otherwise, we split each B and C into two

subsets B^-, B^+ and C^-, C^+. Therefore, computing the base monomial set of CrossBaseMonomials(B^-, C^-) $\cup$ CrossBaseMonomials(B^+, C^+) and CrossBaseMonomials(B^-, C^+) $\cup$ CrossBaseMonomials(B^+, C^-) can be performed in parallel and recursively with an auxiliary function HalfCrossBaseMonomials of Algorithm 12. The results of these two concurrent calls to Algorithm 12 are then merged with Algorithm 8.

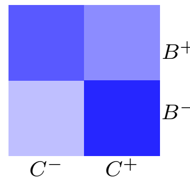


Figure 6.4: Illustration of Algorithm 11 CrossBaseMonomials

- Algorithm 12 simply performs two concurrent calls to Algorithm 11 whose results are merged with Algorithm 8.

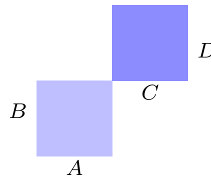


Figure 6.5: Illustration of Algorithm 12 HalfCrossBaseMonomials

To merge two sets of minimal elements, we rely on the procedure ParallelMinMerge (Algorithm 8). We now specify the operations on monomials that are required by Algorithm 8.

- A monomial represented by an exponent vector can be marked by setting its first slot to -1.
- The function “merge” is not needed in this case. If a monomial can be divided by another, then its information is not needed in the base monomial set anymore.
- A set of monomials can be split by partitioning the array representing it to two halves.
- The union of two sets of disjoint monomials can be computed by concatenating the two arrays representing them.

```
Input  : a monomial set A
Output : Min(G_A) where G_A consists of all non-constant gcd(a_1, a_2)
         for a_1, a_2 ∈ A and a_1 ≠ a_2
1  if |A| ≤ MIN_BASE then
2    return NaiveBaseMonomialSet(A);
3  else
4    (A⁻, A⁺) ← Split(A);
5    B ← spawn SelfBaseMonomials(A⁻, A⁺);
6    C ← spawn CrossBaseMonomials(A⁻, A⁺);
7    sync;
8    (D_1, D_2) ← ParallelMinMerge(B, C);
9    return Union(D_1, D_2);
```

Algorithm 9: ParallelBaseMonomials

```
Input  : two disjoint monomial sets B, C
Output : Min(G_B ∪ G_C) where G_B (resp. G_C) consists of all
         non-constant gcd(x, y) for x, y ∈ B (resp. C) with x ≠ y
1  E ← spawn ParallelBaseMonomials(B);
2  F ← spawn ParallelBaseMonomials(C);
3  sync;
4  (D_1, D_2) ← ParallelMinMerge(E, F);
5  return Union(D_1, D_2);
```

Algorithm 10: SelfBaseMonomials

```
Input  : two disjoint monomial sets B, C where B and C differ at
         most 1
Output : Min(G_{B,C}) where G_{B,C} consists of all non-constant
         gcd(b, c) for (b, c) ∈ B × C
1  if |B| ≤ MIN_MERGE_BASE then
2    return SerialCrossBaseMonomials(B, C);
3  else
4    (B⁻, B⁺) ← Split(B);
5    (C⁻, C⁺) ← Split(C);
6    E ← spawn HalfCrossBaseMonomials(B⁻, C⁻, B⁺, C⁺);
7    F ← spawn HalfCrossBaseMonomials(B⁻, C⁺, B⁺, C⁻);
8    sync;
9    (D_1, D_2) ← ParallelMinMerge(E, F);
10   return Union(D_1, D_2);
```

Algorithm 11: CrossBaseMonomials

```
Input   : four monomial sets A, B, C, D pairwise disjoint
Output  : Min(G_{A,B} ∪ G_{C,D}) where G_{A,B} (resp. G_{C,D}) consists of all
          non-constant gcd(x, y) for (x, y) ∈ A × B (resp. C × D)
1 E ← spawn CrossBaseMonomials(A, B);
2 F ← spawn CrossBaseMonomials(C, D);
3 sync;
4 (G_1, G_2) ← ParallelMinMerge(E, F);
5 return Union(G_1, G_2);
```

Algorithm 12: HalfCrossBaseMonomials

```
Input   : two disjoint monomial sets B, C
Output  : Min(G_{B,C}) where G_{B,C} consists of all non-constant
          gcd(b, c) for (b, c) ∈ B × C
1  M ← ∅;
2  for i ← 1 to |B| do
3      for j ← 1 to |C| do
4          g ← Gcd(B_i, C_j); is_min ← true;
5          for m ∈ M do
6              if m | g then
7                  is_min ← false;
8                  break;
9              else if g | m then
10                 M ← M \ {m};
11         if is_min = true then
12             M ← M ∪ {g};
13 return M;
```

Algorithm 13: SerialCrossBaseMonomials

At the end of this subsection, we discuss some implementation details.

- Both in Algorithm 4 and Algorithm 13, in the worst case, to compare each pair g and m, we may traverse these two pairs twice. In the implementation, we are actually testing their divisibility by the function Compare which implements the partial order used on the monomials. In this way, we traverse the two arrays representing these monomials no more than once.

- In Algorithm 11, to test if the input is in base case, we only compare $|B|$ with the threshold. Different from Algorithm 8, we do not do case discuss on the size of B and C. This is due to the fact that we are calling this function initially

with A^- and A^+ whose sizes differ at most 1. This function is also called in Algorithm 12 where the four input are initially with size $\lfloor |A|^-/2 \rfloor$, $\lceil |A|^-/2 \rceil$, $\lfloor |A|^+/2 \rfloor$, $\lceil |A|^+/2 \rceil$. The call to Algorithm 11 in Algorithm 12 also maintains the property that the sizes of the two inputs differ at most 1 ensured by the following equations,

$$\begin{aligned} |a-b| \le 1 &\Rightarrow |\lceil a/2 \rceil - \lfloor b/2 \rfloor| \le 1; \\ |a-b| \le 1 &\Rightarrow |\lceil b/2 \rceil - \lfloor a/2 \rfloor| \le 1, \end{aligned}$$

where $a, b > 0 \in Z$.

- The Algorithm 12 could have been integrated to Algorithm 11. Algebraically, it makes no difference. However, we make it a separate function considering the locality. We expect that once a function is spawned with an input data set, it will work on these data as much as possible without interchanging with other processors.

6.2.3 Construction of the Hypergraph

The Algorithm 2 introduced in Chapter 4 to construct the hypergraph $\mathsf{HG}(f, \mathcal{M})$ proceeds in a very similar manner of Algorithm 4. While the algebraic complexity stays the same, we propose again a divide-and-conquer algorithm to do the same job but taking advantage of the parallelism and data locality. The principle of Algorithm 14 is similar to that of Algorithm 9. However, there are two differences to be pointed out, namely

- When the input size is larger than the threshold, we only split the monomial set of the input polynomial but not the base monomial set, as we expect this base monomial set to be small.

- To merge two hypergraphs sharing the same vertices set, the operations to perform on their hyperedges are different from those on monomials in Algorithm 9. If hyperedge e_i shares its defining monomial with hyperedge e_j, then e_j is replaced by its union with e_i and only e_j stays in the merged hypergraph.

The top level Algorithm 14 uses threshold as well. When the input fits into cache, the computation is done serially. Otherwise, we split $M = \mathsf{monoms}(f)$ into two even parts and create two independent computational branches on each of them. These two concurrent calls return two set of hyperedges $\mathcal{E}^-$ and $\mathcal{E}^+$. Each of these two sets is

"clean" in the sense that there are no hyperedges in them that share the same defining monomials. However, it is possible that there exists $e_1 \in \mathcal{E}^-$ and $e_2 \in \mathcal{E}^+$ that e_1 shares defining monomial with e_2. Therefore we merge $\mathcal{E}^-$ with $\mathcal{E}^+$ with Algorithm 8. The relation $\preceq$ on hyperedges is defined as

$$e_i \preceq e_j \text{ whenever defining monomials of } e_i \text{ and } e_j \text{ are the same.}$$

Note that this relation is not antisymmetric. Our Algorithm 7 and Algorithm 8 actually work in a more general structure than partially ordered set.

```
Input  : a set of monomials M, the base monomial set 𝓜
Output : ℰ, which is the set of hyperedges of hypergraph HG(f, 𝓜)
         where monoms(f) = M
1 if |M| ≤ HG_BASE then
2 |  return NaiveConstructHypergraph(M, 𝓜);
3 else
4 |  (M⁻, M⁺) ← Split(M);
5 |  ℰ⁻ ← spawn ParallelConstructHypergraph(M⁻, 𝓜);
6 |  ℰ⁺ ← spawn ParallelConstructHypergraph(M⁺, 𝓜);
7 |  sync;
8 |  (E⁻, E⁺) ← ParallelMinMerge(ℰ⁻, ℰ⁺);
9 |  return Union(E⁻, E⁺);
```

Algorithm 14: ParallelConstructHypergraph

We present here the operations on hyperedges that are necessary to support Algorithm 8.

- A hyperedge can be marked by setting the first slot of the exponent vector representing its defining monomial to -1.
- When a hyperedge e_i shares its defining monomial with another hyperedge e_j, these two hyperedges will be merged by a function "merge" on hyperedges. It overwrites e_j by its union with e_i.
- A set of hyperedges can be split by partitioning the array representing it to two halves.
- The union of two sets of disjoint hyperedges can be computed by appending the array representing one of them to another.

6.2.4 Computation of the Largest Intersection of Hyperedges

Another procedure that can be parallelized in Algorithm 3 is the computation of the largest intersection of all pairs of hyperedges. To make the story complete, we first state a naive serial way to do this by Algorithm 15. This algorithm will also serve as a base routine of the parallel algorithm (Algorithm 16). The parallel algorithm proceed in a divide-and-conquer manner. It uses also a threshold (INTER_BASE) to control parallel overhead while ensures that the base case computation could fit in cache. When the input size grows, it creates two balanced independent recursive calls which therefore run concurrently. To merge the results of these two recursive functions, we simply leave the larger intersection as output.

```
Input  : a set of hyperedges ℰ
Output : a hyperedge intersection of maximum cardinality
1  I ← ∅ ;
2  for i ← 1 to |ℰ| do
3      for j ← i + 1 to |ℰ| do
4          I′ ← Intersection(ℰi, ℰj);
5          if |I| < |I′| then
6              I ← I′;
7  return I;
```

Algorithm 15: NaiveIntersection

```
Input  : a set of hyperedges ℰ
Output : a hyperedge intersection of maximum cardinality
1  if |ℰ| ≤ INTER_BASE then
2      return NaiveIntersection(ℰ);
3  else
4      (ℰ⁻, ℰ⁺) ← Split(ℰ);
5      I⁻ ← spawn ParallelIntersection(ℰ⁻);
6      I⁺ ← spawn ParallelIntersection(ℰ⁺);
7      sync;
8      if |I⁻| < |I⁺| then
9          return I⁺;
10     else return I⁻;
```

Algorithm 16: ParallelIntersection

6.2.5 Identification of Largest Hyperedge

Similar to the computation of the largest intersection, we can also parallelly identify the largest hyperedge in a given set of hyperedges. We recursively break the problem into sub-problems until they becomes small enough to be solved serially. The base case size is controlled by a threshold LARGE_EDGE_BASE. We will state the algorithm without further discussion on it as the principle is the same as Algorithm 15 and Algorithm 16.

```
   Input  : a set of hyperedges 𝓔
   Output : a hyperedge e of maximum cardinality
 1 if |𝓔| ≤ LARGE_EDGE_BASE then
 2 |  e ← ∅ ;
 3 |  for i ← 1 to |𝓔| do
 4 |  |  if |e| < |𝓔_i| then
 5 |  |  |  e ← 𝓔_i;
 6 |  return e;
 7 else
 8 |  (𝓔⁻, 𝓔⁺) ← Split(𝓔);
 9 |  e⁻ ← spawn ParallelLargestEdge(𝓔⁻);
10 |  e⁺ ← spawn ParallelLargestEdge(𝓔⁺);
11 |  sync;
12 |  if |e⁻| < |e⁺| then
13 |  |  return e⁺;
14 |  else
15 |  |  return e⁻;
```

Algorithm 17: ParallelLargestEdge

6.2.6 Collecting Hyperedges Including the Largest Intersection

Once the largest intersection of two hyperedges (computed by Algorithm 16 in p.60) is found, we traverse the hypergraph to test if it is a subset of more hyperedges. We collecting the defining monomials of all these hyperedges in a set Q and this set forms the monomial set of the cofactor of the candidate syntactic factorization. (Line 14 to Line 16 of Algorithm 3). This procedure can be parallelized via a divide-and-conquer strategy as well. We show the straight forward algorithm without further explanation (the principle is also the same as Algorithm 15 and Algorithm 16).

Input : a set of hyperedges $\mathcal{E}$ and an intersection I of hyperedges
Output : $\mathcal{E}_{sup}$ such that $\{E \in \mathcal{E}_{sup} \mid E \in \mathcal{E}, I \subset E\}$

```
1  if |E| ≤ LARGE_EDGE_BASE then
2      E_sup ← ∅ ;
3      for i ← 1 to |E| do
4          if I ⊂ E_i then
5              E_sup ← Union(E_sup, {E_i});
6      return E;
7  else
8      (E^-, E^+) ← Split(E);
9      E^-_sup ← spawn ParallelSuperSet(E^-);
10     E^+_sup ← spawn ParallelSuperSet(E^+);
11     sync;
12     return Union(E^-_sup, E^+_sup);
```

Algorithm 18: ParallelSuperSet

6.2.7 Multiplication of Two Sets of Monomials

Once the monomial sets of the base and cofactor of a candidate syntactic factorization have been established (set M and set Q in Line 18 in Algorithm 3), We need to compute the pairwise product of monomials in these two sets, that is, given a monomial set M and monomial set Q, compute the set N such that

$$N = \{mq \mid m \in M, q \in Q\}.$$

This set N will only be useful when there is no combination of monomials, i.e.

$$|N| = |M| \cdot |Q|.$$

Therefore, our algorithm could work in the following manner. Until there is no combination of monomials, compute the set N. To compute it, serially, we can compute each product of monomials from the set M and Q and add it to the current set N. If there is no combination of monomials, then continue; otherwise we break the computation. This serial procedure is presented by Algorithm 19 NaiveMulMonomials. Parallelly, we would apply the very similar strategy as procedure ParallelConstructHypergraph (Algorithm 14). As the base monomial set is expected to be small, we do not split it when the input size is large. The difference is how

```
Input   : two monomial sets M, Q
Output  : N = {mq | m ∈ M, q ∈ Q}. If |N| = |M| · |Q|, return
          (true, N); otherwise return (false, ∅).
1 N ← ∅;
2 for i ← 1 to |M| do
3     for j ← 1 to |Q| do
4         m ← M_i · Q_j;
5         for m' ∈ N do
6             if m = m' then
7                 return (false, ∅);
8         N ← N ∪ {m};
9 return (true, N);
```

Algorithm 19: NaiveMulMonomials

```
Input   : two monomial sets M, Q
Output  : N = {mq | m ∈ M, q ∈ Q}. If |N| = |M| · |Q|, return
          (true, N); otherwise return (false, ∅).
1  if |Q| ≤ MUL_MONOMIAL_BASE then
2      return NaiveMulMonomials(M, Q);
3  else
4      (Q⁻, Q⁺) ← Split(Q);
5      (b⁻, N⁻) ← MulMonomials(M, Q⁻);
6      (b⁺, N⁺) ← MulMonomials(M, Q⁺);
7      if b⁻ = b⁺ = true then
8          return MulMerge(N⁻, N⁺);
9      else
10         return (false, ∅);
```

Algorithm 20: MulMonomials

Input : two monomial sets N_1, N_2
Output : if $N_1 \cap N_2 = \emptyset$, return (true, $N_1 \cup N_2$); otherwise return (false, $\emptyset$).

1 **if** $|N_1| \leq$ MUL_Merge_BASE *and* $|N_2| \leq$ MUL_Merge_BASE **then**
2   **return** NaiveMulMerge(N_1, N_2);
3 **else if** $|N_1| >$ MUL_Merge_BASE *and* $|N_2| >$ MUL_Merge_BASE **then**
4   $(N_1^-, N_1^+) \leftarrow$ Split(N_1);
5   $(N_2^-, N_2^+) \leftarrow$ Split(N_2);
6   $(b_1^-, S_1^-) \leftarrow$ **spawn** MulMerge(N_1^-, N_2^-);
7   $(b_1^+, S_1^+) \leftarrow$ **spawn** MulMerge(N_1^-, N_2^+);
8   $(b_2^-, S_2^-) \leftarrow$ **spawn** MulMerge(N_1^+, N_2^-);
9   $(b_2^+, S_2^+) \leftarrow$ **spawn** MulMerge(N_1^+, N_2^+);
10   **sync**;
11   **if** $b_1^- = b_1^+ = b_2^- = b_2^+ =$ *true* **then**
12   **return** Union(Union(S_1^-, S_1^+), Union(S_2^-, S_2^+));
13   **else**
14   **return** (false, $\emptyset$);
15 **else if** $|N_1| >$ MUL_Merge_BASE *and* $|N_2| \leq$ MUL_Merge_BASE **then**
16   $(N_1^-, N_1^+) \leftarrow$ Split(N_1);
17   $(b^-, S^-) \leftarrow$ **spawn** MulMerge(N_1^-, N_2);
18   $(b^+, S^+) \leftarrow$ **spawn** MulMerge(N_1^+, N_2);
19   **sync**;
20   **if** $b^- = b^+ =$ *true* **then**
21   **return** Union(S^-, S^+);
22   **else**
23   **return** (false, $\emptyset$);
24 **else**
25   $(N_2^-, N_2^+) \leftarrow$ Split(N_2);
26   $(b^-, S^-) \leftarrow$ **spawn** MulMerge(N_1, N_2^-);
27   $(b^+, S^+) \leftarrow$ **spawn** MulMerge(N_1, N_2^+);
28   **sync**;
29   **if** $b^- = b^+ =$ *true* **then**
30   **return** Union(S^-, S^+);
31   **else**
32   **return** (false, $\emptyset$);

Algorithm 21: MulMerge

```
    Input  : two monomial sets N_1, N_2
    Output : if N_1 ∩ N_2 = ∅, return (true, N_1 ∪ N_2); otherwise return
             (false, ∅).
1 for i ← 1 to |N_1| do
2     for j ← 1 to |N_2| do
3         if N_1_i = N_2_j then
4             return (false, ∅);
5 return (true, Union(N_1, N_2));
```

Algorithm 22: NaiveMulMerge

the results of two recursive calls are merged. As we would return (false, $\emptyset$) immediately after combination of monomials is detected, we first check if this happens before merging the two product monomial sets. Due to the same reason, more parallelism is allowed to be introduced to procedure MulMerge in Algorithm 21 than procedure ParallelMinMerge in Algorithm 8. The objective of Algorithm 21 may be stated as computing the union of two input monomial sets if they are disjoint. On the other hand, if we want to compute the union of two sets no matter they are disjoint or not, we should call Algorithm 8 with the relation $\preceq$ defined as equality test. As a result, the four recursive calls in Line 6 to Line 9 are now performed concurrently. In the case when one set is small and the other one is large, the two recursive calls (Line 17, 18 and Line 26, 27) are also performed parallelly. This leads to larger parallelism than ParallelMinMerge (Algorithm 8).

6.2.8 Solving Coefficients of a Candidate Syntactic Factorization

We discuss the parallelism of the procedure to compute or reject a candidate syntactic factorization in this subsection. Its serial version has been presented by Proposition 6. We will follow the same notation used in the proposition. Now we are given the set G and set H satisfying the assumption of Proposition 6, the next thing to do is finding the polynomial p such that it may admit a syntactic factorization. To set up the system mentioned in Proposition 6, we do a binary search to locate $|G|\cdot|H|$ monomials in monoms(f). These binary search can be performed concurrently on each of these $|G| \cdot |H|$ monomials. Once the system is set up, as described in Proposition 6, one can freely set g_1 to 1 since the coefficients are in a field. After that, we can parallelly deduce $h_1, \ldots, h_b$. By h_1, parallelly to each i, we can then obtain g_i and then check if

$g_i\, h_j$ lead to a solution for all j. However, the parallel version has not been included in the current implementation and our benchmarks are done without this parallelization. Therefore, we only mention the possible parallelism without stating detail algorithms.

6.2.9 Updating Hypergraph by a Syntactic Factorization

Once a syntactic factorization has been established, the hypergraph should be updated accordingly. We state this problem formally as the following. Given a set of monomials N and a set of hyperedges $\mathcal{E}$, we wish to remove from each hyperedge those elements that after multiplying with its defining monomial result a monomial in set N. That is, for each hyperedge $E_q = \{m_1, m_2, \ldots, m_k\}$, we want to remove from it $m_i, i = 1, \ldots, k$ such that $m_i\, q \in N$. Serially, it can be done via a straight forward algorithm stated by Algorithm 24. The parallel algorithm can be obtained by simply applying the divide-and-conquer strategy. We state it by Algorithm 23. An interesting feature of this parallel algorithm is that it does not require to merge the results of the recursive calls.

We mention one implementation technique of the operation that removes one monomial from a hyperedge in Line 5 of Algorithm 24. Theoretically, there should be no data race if two threads concurrently remove two different monomials from the hyperedge. However, recall that the collection of monomials in a hyperedge is encoded by a bitvector. Thus to remove a monomial from a hyperedge we unset its corresponding bit in the bitvector. This may cause data race on the array encoded bitvector. For example, if two monomials correspond to two bits that reside in the same word and two independent threads are trying to remove these two monomials respectively. These two threads are going to write to the same word concurrently. Due to this reason, our class bitvector supports two types of operation unset, the common serial version and a thread-safe version. The thread safe version is implemented by atomic operation such that before modifying a certain bit, a thread locks the word where this bit resides. The lock is released when the thread finishes modifying the word.

6.2.10 Syntactic Decomposition

For the current implementation, the parallelization of Algorithm 6 SyntacticDecomposition is simply realized by making the for loop at Line 6 a parallel for loop. This parallelization is straight forward as all the syntactic factorizations (g_i, h_i) for $i = 1, 2, \ldots, e$ are independent from each other. Thus the computation on each syntactic factoriza-

```
Input  : a set of monomials N, a set of hyperedges ℰ
Output : for each E_q ∈ ℰ, E_q ← E_q \ {m ∈ E_q | mq ∈ N}

 1 if |N| ≤ UPDATE_BASE and |ℰ| ≤ UPDATE_BASE then
 2     return NaiveUpdate(N, ℰ);
 3 else if |N| > UPDATE_BASE and |ℰ| > UPDATE_BASE then
 4     (N⁻, N⁺) ← Split(N);
 5     (ℰ⁻, ℰ⁺) ← Split(ℰ);
 6     spawn Update(N⁻, ℰ⁻);
 7     spawn Update(N⁻, ℰ⁺);
 8     spawn Update(N⁺, ℰ⁻);
 9     spawn Update(N⁺, ℰ⁺);
10     sync;
11 else if |N| > UPDATE_BASE and |ℰ| ≤ UPDATE_BASE then
12     (N⁻, N⁺) ← Split(N);
13     spawn Update(N⁻, ℰ);
14     spawn Update(N⁺, ℰ);
15     sync;
16 else
17     (ℰ⁻, ℰ⁺) ← Split(ℰ);
18     spawn Update(N, ℰ⁻);
19     spawn Update(N, ℰ⁺);
20     sync;
```

Algorithm 23: Update

```
Input  : a set of monomials N, a set of hyperedges ℰ
Output : for each E_q ∈ ℰ, E_q ← E_q \ {m ∈ E_q | mq ∈ N}

1 for i ← 1 to |N| do
2     for j ← 1 to |ℰ| do
3         q ← defining monomial of ℰ_j;
4         if q | N_i then
5             ℰ_j ← ℰ_j \ {N_i/q};
```

Algorithm 24: NaiveUpdate

tions can be made concurrent. Ideally, we also wish to make this parallelization a divide-and-conquer algorithm such that the cache locality could be exploited better. However, to split a set of syntactic factorizations evenly in the sense that the work intended to be done on the two resulting halves are balanced requires much more care than splitting a set of monomials or hyperedges. Furthermore, the base case should not only been decided by the cardinality of the set. Indeed, as we would like the base case computation fits into cache while performs enough work to reduce parallel overhead, the parameters to decide a base case should contain at least the number of terms, the degree of each (g_i, h_i) and of course the number of syntactic factorizations. For now we simply spawn all syntactic factorizations and rely on the `Cilk++` scheduler to balance the work between processors. Further research needs to be done on the divide-and-conquer parallelization of this top level algorithm.

Chapter 7

Evaluation Scheduling

In this chapter, we describe how to statically compute a schedule for efficient parallel evaluation of a syntactic decomposition. Recall that, by definition, a syntactic decomposition is a binary tree whose internal nodes are operators $+, -, \times$ and whose leaves belong to $\mathbb{K} \cup X$. As mentioned in (Subsection 6.1.3 p. 47), at implementation level, a syntactic decomposition is represented by the data structure *SynDecomposition*. With procedure ExpressionTree (Algorithm 5 p. 39) at hand, following the Formula (6.1) (p. 49), we recursively convert a syntactic decomposition encoded by a *SynDecomposition* object to a binary tree, represented by an SLP (data structure presented in Section 2.2).

After eliminating common subexpressions (for instance by Algorithm 1 p. 18), a binary tree becomes a directed acyclic graph (DAG) which can be evaluated in its topological ordering. However, targeting multi-core platforms, our objective is to decompose a DAG into p sub-DAGs for a given parameter p, the number of available processors. The requirement on these sub-DAGs is that the evaluation of one sub-DAG does not depend on the evaluation of the other. which guarantees the correctness of DAG evaluation on multi-processors. We also expect that these sub-DAGs to be balanced in size such that the "span" of the intended parallel evaluation is minimized.

Roughly, a schedule, a finite set of sub-DAGs, is generated in two steps. We first collect all common subexpressions, and treat them as leaves. Then the problem of scheduling DAGs reduces to that of scheduling binary trees, where broadcasting common subexpressions may be necessary. We explain this with the following example.

Example 8. *The DAG on the left in Figure 7.1 represents the polynomial*

$$abc(d+e) + f(abc+h),$$

where the shaded nodes represent the common subexpression abc. We evaluate this

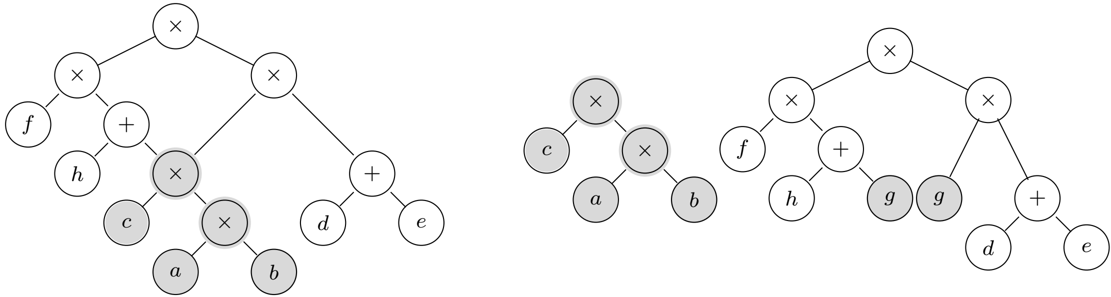


Figure 7.1: Evaluate a DAG in two steps

DAG in two steps. First the common subexpression abc is separated. Then the evaluation of the binary tree on the right is considered, where the common subexpression abc is treaded as an input leaf labelled as g. This may also be understood as adding an additional line $g = abc$ into the equivalent SLP of the input DAG.

From now on, we concentrate on the following question. Let T be a binary tree. Given a fixed parameter p, the number of subprograms intended, we describe a way to partition T into p groups of subtrees of T such that each group can be evaluated independently. More precisely, $\mathcal{G} = \{G_1, \ldots, G_p\}$ is p-schedule of T if

(1) G_i is a set of maximal subtrees of T for all i,

(2) no node in G_i appears in G_j for all $i \neq j$,

where a subtree of T is maximal if it contains all the children of its root node. Before presenting the algorithm in detail, we introduce some notations.

Definition 12 (the load of a node)**.** *We assume that there are two constants,* ADD *and* MULT *representing the cost to perform an addition or an multiplication in the field $\mathbb{K}$ respectively. For each node t in a binary tree T, we associate to it a cost* $\mathsf{load}(t)$ *as follows.*

- *If t is an element in $\mathbb{K}$ or a variable in X, then the load of t is defined as 0;*
- *if t is an addition node with first operand t_1 and second operand t_2, then the load of t is the sum of the load of these two operands plus an addition cost, written as*

$$\mathsf{load}(t) = \mathsf{load}(t_1) + \mathsf{load}(t_2) + \mathsf{ADD};$$

- *if t is a multiplication node with first operand t_1 and second operand t_2, then the load of t is the sum of the load of these two operands plus an multiplication cost, written as*
$$\mathsf{load}(t) = \mathsf{load}(t_1) + \mathsf{load}(t_2) + \mathsf{MULT}.$$

The load of a maximal subtree G of a binary tree T, denoted by $\mathsf{load}(G)$, is defined as the load of its root. Note that T is a maximal subtree of itself. The load of all nodes can be computed by traversing the binary tree once, illustrated by the following example.

Example 9. *We compute the load of each node in the binary tree in Figure 7.1. We assume we are in a field where the cost to perform an addition or multiplication can be counted as unit cost. The load of each node is marked inside the node in the right part of Figure 7.2.*

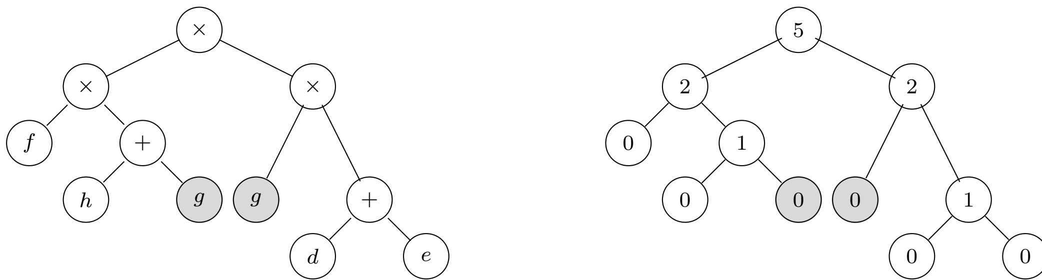


Figure 7.2: The load of nodes in a binary tree.

The load of a set of maximal subtrees is defined as the sum of the load of each subtree. More precisely, let $G = \{T_1, T_2, \ldots, T_k\}$, where each T_i for $i = 1, \ldots, k$ is a maximal subtree. The load of G is defined as

$$\mathsf{load}(G) = \sum_{1 \le i \le k} \mathsf{load}(T_i).$$

More generally, the load of a p-schedule $\mathcal{G} = \{G_1, \ldots, G_p\}$ of a binary tree T is defined as

$$\mathsf{load}(\mathcal{G}) = \sum_{1 \le i \le p} \mathsf{load}(G_i).$$

Note that there are usually some nodes of T which do not belong to any subtree in the p-schedule $\mathcal{G}$, and they may be regarded as synchronization nodes. Clearly, the inequality $\mathsf{load}(\mathcal{G}) \le \mathsf{load}(T)$ holds.

Given a scheduling $\mathcal{G}$ of a binary tree T, the objective is to minimize the span of $\mathcal{G}$ with respect to T, defined as

$$\mathsf{span}(\mathcal{G}) = \max_{1 \le i \le p} \mathsf{load}(G_i) \; + \; \mathsf{load}(T) - \mathsf{load}(\mathcal{G}) \tag{7.1}$$

Formula (7.1) can be understood as the following. To evaluate a binary tree T using p processors, given a p-schedule $\mathcal{G} = \{G_1, \ldots, G_p\}$ of T, we assign each G_i for $i = 1, \ldots, p$ to one of p processors. These evaluation can be done concurrently. After all the nodes in the p-schedule $\mathcal{G}$ have been evaluated, we sequentially evaluate the rest nodes of T that are not included in $\mathcal{G}$.

Assume that each node t in T is associated with its load. Now we describe how to obtain a p-schedule of T.

By choosing a cut-off value $K > 0$, with a tree traversal, one can find the set M of the maximal subtrees of T such that each root of the maximal subtree has load at most K (Algorithm 26). Then we construct the set

$$L = \{\mathsf{load}(m) \mid m \in M\}$$

(assume that subtrees have different load, otherwise use a multi-set instead). It is not hard to see that a partition of L with size p produces a p-schedule of T. However, finding a balanced partition of L is hard, and this is the well-known multiprocessor scheduling problem, which is an NP-complete problem. Fortunately, the simple LPT-algorithm (Longest Processing Time) achieves a partition within a factor 4/3 of optimal [29], and the formal algorithm to produce a p-schedule of a binary tree is given by Algorithm 25 and Algorithm 26.

In summary, to minimize $\mathsf{span}(\mathcal{G})$, one need to balance the load among all G_i's while keeping $\mathsf{load}(\mathcal{G})$ close to $\mathsf{load}(T)$. We set the cut-off value $K = \lceil \mathsf{load}(T) \,/\, p \rceil$, the average number of nodes in a group, and expect this cut-off value works well in practice as observed in our experimentation.

Example 10. *Consider a binary tree T with load 615 as in Figure 7.3. For $p = 4$, the cut-off value for this binary tree is $\lceil \mathsf{load}(T) \,/\, p \rceil = \lceil 615 \,/\, 4 \rceil = 154$. The load of the roots of maximal subtrees discovered are*

$$L = \{22, 80, 126, 29, 52, 13, 101, 121, 18, 44\},$$

which can be partitioned, by the LPT algorithm, into four groups $\{126, 22\}$, $\{121, 29\}$, $\{101, 44, 13\}$, $\{80, 52, 18\}$ with load 148, 150, 158, 150, respectively. The total load

Input : a binary tree T, the number of intended sub-trees p
Output : a p-schedule of T

```
1  K ← ⌈load(T)/p⌉;
2  M ← MaximalTreeRoots(T, K);
3  for i ← 1 to p do
4  |   G_i ← ∅;
5  |   load(G_i) ← 0;
6  sort M decreasingly by the load of each node;
7  for i ← 1 to |M| do
8  |   find j such that G_j has the minimal load;
9  |   add the maximal tree with root M_i to G_j;
10 |   load(G_j) ← load(G_j) + load(M_i) ;
11 return {G_1, G_2, ..., G_p};
```

Algorithm 25: ScheduleBinaryTree

Input : a binary tree T, a cut-off value K
Output : a set of nodes each of which as the root of a maximal subtree has load less than K

```
1  if load(T) ≤ K then
2  |   return {Root(T)};
3  else
4  |   M_ℓ ← MaximalTreeRoots(T_ℓ);
5  |   M_r ← MaximalTreeRoots(T_r);
6  |   return M_ℓ ∪ M_r;
```

Algorithm 26: MaximalTreeRoots

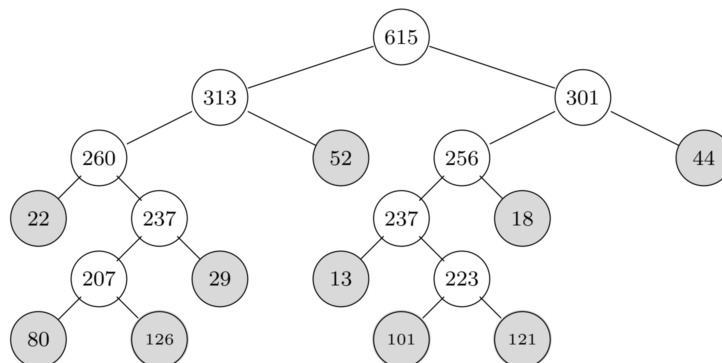


Figure 7.3: A portion of a binary tree with 615 nodes. Each shaded node is the root of a maximal subtree, with children omitted.

of nodes which do not appear in any subtrees is 9*, and the span of this schedule is* $158 + 9 = 167$*.*

As of implementation, given an input DAG, we first evaluate its common subexpressions and then evaluate the p-schedule of the resulting binary tree. Thus after a p-schedule has been computed, we reorder the array representing the input DAG such that common subexpressions appear together, followed by p groups of nodes. This reordering allows cache-efficient evaluations of the DAG and potentially reduces false sharings among different processors.

Chapter 8

Experimentation

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 [13], the `optimize` function with `tryhard` option provided by the computer algebra system Maple and our algorithm presented in Chapter 4 and Chapter 6. As described in the introduction, we use the evaluation of resultants of generic polynomials as a driving example. Our input is taken as the resultant of two generic univariate polynomials with degrees ranging from 4 to 8. For larger generic resultant, we are not aware of any software that can finish computing it (mainly due to memory exhaustion). 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.

8.1 Evaluation Cost

After a syntactic decomposition of the input polynomial is obtained, we eliminate common subexpressions in it to further reduce its evaluation cost. As a result, while talking about the output size, there are two columns, one before and one after the common subexpression elimination. We count each arithmetic operation (addition, multiplication) to the same unit cost.

Table 8.1 shows the cost to evaluate the representation of 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 optimized by different approaches. The first two columns of Table 8.1 gives m and n. The third column indicates the number of monomials appearing in $R(a, b)$. The number of arithmetic operations required to evaluate the input polynomial naively term by term, as computed by MAPLE, is given by the fourth col-

umn *Input*. The fifth column *Horner* is the evaluation cost of the polynomial 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. This command has integrated multivariate Horner's scheme, common subexpression elimination technique and some heuristic methods. The last two columns reports the evaluation cost of the syntactic decomposition of the input polynomial computed by Algorithm 6, before and after removing common subexpressions. We notice that in our tested examples, the cost to evaluate a syntactic decomposition can be reduced by at least a half by applying this post-processing technique (for which we use standard techniques running in time linear w.r.t. input size). The typical output of these optimization tools has been shown in Appendix A (p. 104). We note that the evaluation cost of the representation returned by $SD + CSE$ is less than the ones obtained with the Horner's rule and MAPLE's `optimize` function.

m	n	#Mon	Input	Horner	tryhard	SD	SD + CSE
4	4	219	1,876	977	620	899	549
5	4	549	5,199	2,673	1,496	2,211	1,263
5	5	1,696	18,185	7,779	4,056	7,134	3,543
6	4	1,233	13,221	6,539	3,230	4,853	2,547
6	5	4,605	54,269	22,779	10,678	18,861	8,432
6	6	14,869	190,890	69,909	31,760	63,492	24,701
7	4	2,562	30,438	14,948	6,707	9,862	4,905
7	5	11,380	146,988	61,399	27,363	45,546	19,148
7	6	43,166	601,633	219,341	-	179,870	65,770
7	7	145,330	2,166,653	697,743	-	627,584	206,840
8	4	4,970	63,731	19,547	12,191	18,730	8,826
8	5	25,917	359,487	106,800	-	101,327	39,816
8	6	114,080	1,700,662	498,410	-	464,593	157,312
8	7	441,145	7,028,510	2,042,037	-	1,863,653	565,020
8	8	1,524,326	25,838,829	*	-	6,648,972	1,844,464

* means that the computation is killed due to 0% CPU usage and 90% memory usage.
- means that the computation does not terminate after 5 days.

Table 8.1: Cost to evaluate resultants by different approaches

We also generated some examples other than resultants to test the output size of out algorithm. Those input data are generated by randomly taking the product or sum of a certain number of random forms. We expect more sophisticated syntactic factorizations can be extracted from these input polynomials than generic resultants. It has been shown in Table 8.2 that at these input our algorithm performs much better than the other two approaches.

#Mon	Input	Horner	tryhard	SD	SD + CSE
2,000	23,624	12,907	4,079	4,950	1,681
3,125	46,874	26,215	6,320	7,890	3,200
3,750	54,374	30,877	7,929	12,193	4,445
6,245	93,679	42,462	14,147	13,613	6,405
9,065	132,093	55,537	17,375	19,222	9,034
15,121	227,242	106,076	29,709	36,529	16,614

Table 8.2: Cost to evaluation hand made input polynomials

8.2 Timing to Optimize Large Polynomials

Table 8.3 shows the timing in seconds that each approach takes to optimize the polynomials analyzed in Table 8.1. The first three columns of Table 8.3 have the same meaning as in Table 8.1. The columns *Horner*, *tryhard* show the timing of optimizing these polynomials. The last column *SD* shows the timing of our parallelized procedure SyntacticDecomposition to produce the syntactic decompositions with our `Cilk++` implementation running on 1 core. All the sequential benchmarks (*Horner*, *tryhard*) were conducted on a 64bit Intel Quad CPU 2.40 GHZ machine with 4 MB L2 cache and 3 GB main memory. The parallel benchmarks were carried out on a machine with 8 Quad Core AMD Opteron 8354 @ 2.2 GHz connected by 8 sockets. Each core has 64 KB L1 data cache and 512 KB L2 cache. Every four cores share 2 MB of L3 cache. The total memory is 128.0 GB. Note that in Table 8.3, our implementation *SD* is conducted on a different machine with the other two compared approaches *Horner* and *tryhard*. However, running on 1 core, this machine is actually the weaker one. We have tested our program on the same machine (also running on 1 core) with the one on which the other two methods are conducted. For $(m, n) = (7, 6)$, our computation terminates in 273.188 seconds. For $(m, n) = (7, 7)$, it terminates in 2868.011 seconds.

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 larger than degree 7 and 6, it does not terminate after 136 hours. For the input $(7, 6)$, our algorithm completes within 7 minutes on one core. For the largest input $(8, 8)$, even Horner's rule, which is known for its efficiency, would not process it due to memory problem.

We report the parallel time of our algorithm in Table 8.4. The first three columns of Table 8.4 have the same meaning as in Table 8.1. The last six columns shows

m	n	#Mon	Horner	tryhard	SD
4	4	219	0.116	7.776	0.028
5	4	549	0.332	49.207	0.080
5	5	1,696	1.276	868.118	0.617
6	4	1,233	0.988	363.970	0.409
6	5	4,605	4.868	8,658.037	4.820
6	6	14,869	24.378	145,602.915	43.764
7	4	2,562	4.377	1,459.343	2.407
7	5	11,380	24.305	98,225.730	33.156
7	6	43,166	108.035	>5 days	404.708
7	7	145,330	191.184	>5 days	4,252.534
8	4	4,970	3.744	6,528.992	8.497
8	5	25,917	23.858	>5 days	189.259
8	6	114,080	145.385	>5 days	3,240.737
8	7	441,145	930.966	>5 days	45,380.056
8	8	1,524,326	*	>5 days	494,362.097

* means that the computation is killed due to 0% CPU usage and 90% memory usage.

Table 8.3: Timing to optimize large polynomials

the timing of our algorithm running on 1, 2, 4, 8, 16 and 32 cores respectively. On large enough input, our implementation shows a near linear speedup when the number of available processors is within 16. However, the speedup does not scale linearly anymore when the number of processors grows up to 32. For example, on the input $(8, 7)$, we achieve speedup around 19 when using 20 cores. The largest speedup is 26 when 30 cores are available. However, with 32 cores the speedup is also around 26 though we have abundant parallelism (11444) according to Cilkview. In the following Figure 8.1, we also notice that the ideal parallelism and the lower performance bound estimated by Cilkview are very high but our measured speedup curve is lower than the lower performance bound. We attribute this performance degradation to parallel overhead due to unbalanced load between processors. With a project of this size, the data movement from one procedure to another would also contribute to its performance.

We point out the following possible improvement of parallelizing the computation of a syntactic decomposition. The most effective one we believe would be designing a reasonable divide-and-conquer algorithm to parallelize the top level algorithm SyntacticDecompostition (Algorithm 6). For now, our implementation to parallelize Algorithm 6 is simply making the for loop at its Line 6 a parallel for loop. This parallelization is straight forward and should have the same span as a possible divide-and-

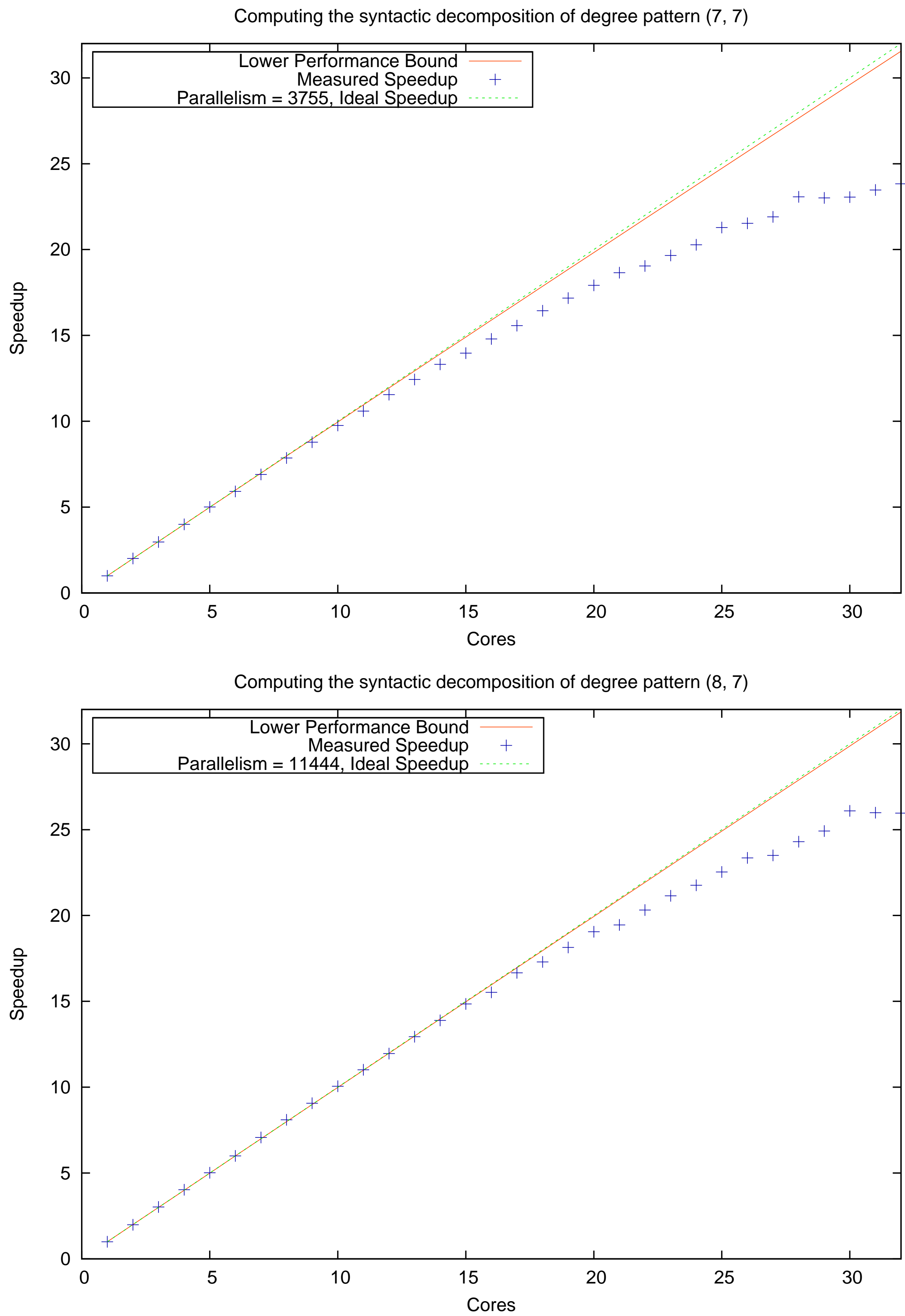


Figure 8.1: Scalability analysis for parallelized SyntacticDecomposition by Cilkview

m	n	#Mon	1-core	2-core	4-core	8-core	16-core	32-core
4	4	219	0.028	0.021	0.015	0.014	0.019	0.036
5	4	549	0.080	0.052	0.037	0.038	0.042	0.067
5	5	1,696	0.617	0.333	0.197	0.164	0.126	0.183
6	4	1,233	0.409	0.237	0.144	0.107	0.107	0.157
6	5	4,605	4.820	2.502	1.498	0.818	0.571	0.597
6	6	14,869	43.764	22.319	11.295	6.013	3.422	2.517
7	4	2,562	2.407	1.119	0.603	0.371	0.308	0.346
7	5	11,380	33.156	16.579	8.629	4.608	2.668	1.974
7	6	43,166	404.708	204.736	105.188	53.336	28.147	16.785
7	7	145,330	4,252.534	2,117.413	1,065.108	540.890	287.396	178.443
8	4	4,970	8.497	4.257	2.289	1.361	0.867	0.773
8	5	25,917	189.259	95.391	48.601	24.968	13.901	8.581
8	6	114,080	3,240.737	1,631.613	817.980	412.707	214.442	126.666
8	7	441,145	45,380.056	22,846.633	11,259.994	5,601.333	2,922.776	1,747.997
8	8	1,524,326	494,362.097	254,547.387	128,589.919	63,356.012	33,261.262	19,804.056

Table 8.4: Speedup of parallel SyntacticDecomposition

conquer realization. However, according to the documentation of Cilk++ ([16]), the compiler executes a parallel for loop by generating a divide-and-conquer recursion. This transformation would in general work well on inputs where simply partitioning the for loop into two halves would create two balanced computational branches. Our case is on the contrary. As Algorithm 3 is developed following greedy strategy, the intended computations on the syntactic factorizations returned by it are highly unbalanced. Relying on the work stealing scheduler of Cilk++ to balance these computations causes parallel overhead and therefore prevents the implementation from achieving linear speedup on cores more than 30. Another possible improvement would be parallelizing the process to solve the system mentioned in Proposition 6. The way to parallelize it has been suggested at Subsection 6.2.8.

8.3 Evaluation Schedule

We report in this section the evaluation schedule computed by Algorithm Schedule from syntactic decompositions of the generic resultants described at Section 8.1. In Table 8.5 and Table 8.6, the columns m and n are the degrees of generic polynomials whose resultants we are taking as input. The column T records the size of a syntactic decomposition after removing of common subexpressions, counting the number of operations. The column T' indicates the number of nodes got scheduled by ScheduleBinaryTree as defined in Chapter 7. The last column indicates the number of operations in each group of the computed 4-schedule and 8-schedule.

For $p = 4$ and 8, we notice that the value $T - T' - \#CS$ is less than 0.5% of

m	n	T	$\#CS$	T'	4-schedule
6	5	8510	1455	7030	1760, 1761, 1755, 1754
6	6	24820	4491	20294	5082, 5069, 5072, 5071
7	5	19293	3169	16073	4029, 4012, 4017, 4015
7	6	66022	11167	54792	13694, 13699, 13717, 13682
7	7	207289	35096	172073	43195, 42981, 42949, 42948
8	5	40051	6812	33186	8305, 8287, 8292, 8302
8	6	157784	28461	129217	32347, 32289, 32281, 32300
8	7	565909	103311	462395	115625, 115589, 115603, 115578
8	8	1846280	345446	1500295	375772, 374969, 374779, 374775

Table 8.5: Parallel evaluation 4-schedule

m	n	T	$\#CS$	T'	8-schedule
6	5	8510	1455	7015	879, 877, 873, 871, 874, 884, 881, 876
6	6	24820	4491	20277	2514, 2579, 2513, 2538, 2538, 2514, 2552, 2529
7	5	19293	3169	16053	2000, 2005, 2002, 2009, 2007, 2002, 2025, 2003
7	6	66022	11167	54773	6846, 6856, 6836, 6836, 6845, 6843, 6850, 6861
7	7	207289	35096	172052	21491, 21578, 21457, 21485, 21449, 21489, 21646, 21457
8	5	40051	6812	33172	4145, 4138, 4144, 4159, 4145, 4153, 4157, 4131
8	6	157784	28461	129198	16140, 16162, 16128, 16159, 16137, 16188, 16148, 16136
8	7	565909	103311	462368	57826, 57949, 57741, 57817, 57742, 57741, 57806, 57746
8	8	1846280	345446	1500270	187461, 187465,187492, 187456, 187513, 187585, 187469, 187829

Table 8.6: Parallel evaluation 8-schedule

T for all the schedules. It indicates that except those common expressions, most of nodes in these syntactic decompositions got scheduled for parallel evaluation. On the other hand, the load of each schedule is close to the other, which indicates that the amount of work assigned to p different processors are balanced. Hence, the method described before works very well for the syntactic decompositions of generic resultants. However, if we serially compute these common subexpressions prior to all the other operations, it would eventually dominant the span of the resulting schedule as the number of processors grows. Therefore, we suggest future work to further discover the dependency between common subexpressions and the other operations and among common subexpressions themselves.

m	n	#point	Input	SD	SD+CSE	4-schedule
6	5	10K	14.490	2.675	1.816	0.997
6	6	10K	57.853	18.618	4.281	2.851
7	5	10K	46.180	11.423	4.053	2.104
7	6	10K	190.397	54.552	13.896	8.479
6	5	10K	6.611	1.241	0.836	0.435

Table 8.7: Timing to evaluate large polynomials at 10K Points

We generated 4-schedules of our syntactic decompositions and compared with three other methods for evaluating our test polynomials on a large number of uni-

m	n	#point	Input	SD	SD+CSE	4-schedule
6	5	100K	144.838	26.681	18.103	9.343
6	6	100K	577.624	185.883	42.788	28.716
7	5	100K	461.981	114.026	40.526	19.560
7	6	100K	1902.813	545.569	138.656	81.270
6	5	100K	66.043	12.377	8.426	4.358

Table 8.8: Timing to evaluate large polynomials at 100K Points

formly generated random points over Z/pZ where $p = 2147483647$ is the largest 31-bit prime number. Our experimental data are summarized in Table 8.7 and Table 8.8. Out the four different evaluation methods, the first three are sequential and are based on the following representations: the original input polynomial $Input$, the syntactic decomposition SD, the common subexpression elimination technique applied syntactic decomposition $SD + CSE$. The last method uses the 4-schedule generated from the representation of the third method $SD + CSE$. 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 Table 8.7 and Table 8.8, we report the timings of the four approaches to evaluate the input at $10K$ and $100K$ points respectively. 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.

Chapter 9

Parallel Computation of the Minimal Elements of a Partially Ordered Set

Inspired by the computation of base monomial set, we are lead to a more general question: how to compute the minimal elements in a partially ordered set? In [43], a joint work with Charles E. Leiserson, Marc Moreno Maza and Yuzhen Xie, we propose a cache-friendly parallel algorithm to solve this general problem and report on its application to the transversal hypergraph generation. This Chapter and the presentation of Algorithm 8 in Chapter 6 are essentially taken from [43].

Partially ordered sets arise in many topics of mathematical sciences. Typically, they are one of the underlying algebraic structures of a more complex entity. For instance, a finite collection of algebraic sets $V = \{V_1, \ldots, V_e\}$ (subsets of some affine space $\mathbb{K}^n$ where $\mathbb{K}$ is an algebraically closed field) naturally forms a partially ordered set (poset, for short) for the set-theoretical inclusion. Removing from V any V_i which is contained in some V_j for $i \neq j$ is an important practical question which simply translates to computing the maximal elements of the poset $(V, \subseteq)$. This simple problem is in fact challenging since testing the inclusion $V_i \subseteq V_j$ may require costly algebraic computations. Therefore, one may want to avoid unnecessary inclusion tests by using an efficient algorithm for computing the maximal elements of the poset $(V, \subseteq)$. However, this problem has received little attention in the literature [15] since the questions attached to algebraic sets (like decomposing polynomial systems) are of much more complex nature.

Another important application of the calculation of the minimal elements of a finite poset is the computation of the transversal of a hypergraph [4, 31], which itself has

numerous applications, like artificial intelligence [20], data mining [30], computational biology [32], mobile communication systems [55], etc. For a given hypergraph $\mathcal{H}$, with vertex set V, the transversal hypergraph $\mathsf{Tr}(\mathcal{H})$ consists of all minimal transversals of $\mathcal{H}$: a transversal $\mathcal{T}$ is a subset of V having nonempty intersection with every hyperedge of $\mathcal{H}$, and is minimal if no proper subset of $\mathcal{T}$ is a transversal. Articles discussing the computation of transversal hypergraphs, as those discussing the removal of the redundant components of an algebraic set generally take for granted the availability of an efficient routine for computing the maximal (or minimal) elements of a finite poset.

Today's parallel hardware architectures (multi-cores, graphics processing units, etc.) and computer memory hierarchies (from processor registers to hard disks via successive cache memories) enforce revisiting many fundamental algorithms which were often designed with *algebraic complexity* as the main complexity measure and with *sequential running time* as the main performance counter. In the case of the computation of the maximal (or minimal) elements of a poset this is, in fact, almost a first visit. Up to our knowledge, there is no published papers dedicated to a general algorithm solving this question. The procedure analyzed in [41] is specialized to posets that are Cartesian products of totally ordered sets.

In this section, we propose an algorithm for computing the minimal elements of an arbitrary finite poset. Our motivation is to obtain an efficient implementation in terms of parallelism and data locality. This divide-and-conquer algorithm, presented in Section 9.1, follows the cache-oblivious philosophy introduced in [23]. Referring to the *multithreaded fork-join parallelism* model of Cilk [24], our algorithm has work $O(n^2)$ and span (or critical path length) $O(n)$, counting the number of comparisons, on an input poset of n elements. A straightforward algorithmic solution with a span of $O(\log(n))$ can be achieved in principle. This algorithm does not, however, take advantage of sparsity in the output, where the discovery that an element is nonminimal allows it to be removed from future comparisons with other elements. Our algorithm eliminates nonminimal elements immediately so that no work is wasted by comparing them with other elements. Moreover, our algorithm does not suffer from determinacy races and can be implemented in Cilk with `sync` as the only synchronization primitive. Experimental results show that our code can reach linear speedup on 32 cores for n large enough.

In several applications, the poset is so large that it is desirable to compute its minimal (or maximal) elements concurrently to the generation of the poset itself, thus avoiding storing the entire poset in memory. Apart from its application to the

computation of base monomial set introduced in Subsection 6.2.2, we illustrate this strategy with its another important applications transversal hypergraph generation in Section 9.4. In both applications, we generate the poset in a divide-and-conquer manner and at the same time we compute its minimal elements. Since, for these two applications, the number of minimal elements is in general much smaller than the poset cardinality, this strategy turns out to be very effective and allows computations that could not be conducted otherwise.

9.1 The Algorithm

In this section, we formally review the notion of a partially ordered set and the notion of minimal elements. Let $\mathcal{X}$ be a set and $\preceq$ be a partial order on $\mathcal{X}$, that is, a binary relation on $\mathcal{X}$ which is reflexive, antisymmetric, and transitive. The pair $(\mathcal{X}, \preceq)$ is called a *partially ordered set*, or poset for short. If A is a subset of $\mathcal{X}$, then $(A, \preceq)$ is the poset induced by $(\mathcal{X}, \preceq)$ on A. When clear from context, we will often write A instead of $(A, \preceq)$. Here are a few examples of posets:

1. $(Z, |)$ where $|$ is the divisibility relation in the ring Z of integer numbers,
2. $(2^S, \subseteq)$ where $\subseteq$ is the inclusion relation in the ranked lattice of all subsets of a given finite set S,
3. $(\mathcal{C}, \subseteq)$ where $\subseteq$ is the inclusion relation for the set $\mathcal{C}$ of all algebraic curves in the affine space of dimension 2 over the field of complex numbers.

An element $x \in \mathcal{X}$ is *minimal* for $(\mathcal{X}, \preceq)$ if for all $y \in \mathcal{X}$ we have: $y \preceq x \Rightarrow y = x$. The set of the elements $x \in \mathcal{X}$ which are minimal for $(\mathcal{X}, \preceq)$ is denoted by $\mathsf{Min}(\mathcal{X}, \preceq)$, or simply $\mathsf{Min}(\mathcal{X})$. From now on we assume that $\mathcal{X}$ is finite.

Algorithms 27 and 28 compute $\mathsf{Min}(\mathcal{X})$ respectively in a sequential and parallel fashion. Algorithm 27 illustrates a straight-forward sequential implementation of the computation of $\mathsf{Min}(A)$, which follows from this trivial observation: an element $a_i \in A$ is minimal for $\preceq$ if for all $j \neq i$ the relation $a_j \preceq a_i$ does not hold. However, Algorithm 27 can not be parallelized while taking advantage of the sparsity in the output. In addition, unless the input data fits in cache, Algorithm 27 is not cache-efficient. We shall return to this point in Section 9.2 where cache complexity estimates are provided.

Algorithm 28 follows the cache-oblivious philosophy introduced in [23]. More precisely, and similarly to the matrix multiplication algorithm of [23], Algorithm 28

```
Input   : a poset A
Output  : Min(A)
1  for i from 0 to |A|−2 do
2      if a_i is unmarked then
3          for j from i+1 to |A|−1 do
4              if a_j is unmarked then
5                  if a_j ⪯ a_i then
6                      mark a_i and break inner loop;
7                  if a_i ⪯ a_j then
8                      mark a_j;
9  A ← {unmarked elements in A};
10 return A;
```

Algorithm 27: SerialMinPoset

```
Input   : a poset A
Output  : Min(A)
1  if |A| ≤ MIN_BASE then
2      return SerialMinPoset(A);
3  (A−, A+) ← Split(A);
4  A− ← spawn ParallelMinPoset(A−);
5  A+ ← spawn ParallelMinPoset(A+);
6  sync;
7  (A−, A+) ← ParallelMinMerge(A−, A+);
8  return Union(A−, A+);
```

Algorithm 28: ParallelMinPoset

proceeds in a divide-and-conquer fashion such that when a subproblem fits into the cache, then all subsequent computations can be performed with no further cache misses. However, Algorithm 28, and other algorithms in this paper, use a threshold such that, when the size of the input is within this threshold, then a base case subroutine is called. In principle, this threshold can be set to the smallest meaningful value, say 1, and thus Algorithm 28 is cache-oblivious. In a software implementation, this threshold should be large enough so as to reduce parallelization overheads and recursive call overheads. Meanwhile, this threshold should be small enough in order to guarantee that, in the base case, cache misses are limited to cold misses. In the implementation of the matrix multiplication algorithm of [23], available in the `Cilk++` distribution, a threshold is used for the same purpose.

In Algorithm 28, when $|A| \leq \mathsf{MIN_BASE}$, where $\mathsf{MIN_BASE}$ is the threshold, Algorithm 27 is called. Otherwise, we partition A into a balanced pair of subsets A^-, A^+. By balanced pair, we mean that the cardinalities $|A^-|$ and $|A^+|$ differ at most by 1. The two recursive calls on A^- and A^+ in Lines 4 and 5 of Algorithm 28 will compare the elements in A^- and A^+ separately. Thus, they can be executed in parallel and free of data races. In Lines 4 and 5 we overwrite each input subset with the corresponding output one so that at Line 6 we have $A^- = \mathsf{Min}(A^-)$ and $A^+ = \mathsf{Min}(A^+)$. Line 6 is a synchronization point which ensures that the computations in Lines 4 and 5 complete before Line 7 is executed. At Line 7, cross-comparisons between A^- and A^+ are made, by means of the operation ParallelMinMerge of Algorithm 8.

9.2 Complexity Analysis and Experimentation

We shall establish a worst case complexity for the work, the span of Algorithm 28 and the cache complexity of its C elision. More precisely, we assume that the input poset of this algorithm has $n \geq 1$ elements, which are pairwise incomparable for $\preceq$, that is, neither $x \preceq y$ nor $y \preceq x$ holds for all $x \neq y$. Our running time is estimated by counting the number of comparisons, that is, the number of times that the operation $\preceq$ is invoked. The costs of all other operations are neglected. The principle of Algorithm 28 is similar to that of a parallel merge-sort algorithm with a parallel merge subroutine, which might suggest that the analysis is standard. The use of thresholds requires, however, a bit of care.

We introduce some notations. For Algorithms 27 and 28 the size of the input is $|A|$ whereas for Algorithms 8 and 7 the size of the input is $|B| + |C|$. We denote by $W_1(n)$, $W_2(n)$, $W_3(n)$ and $W_4(n)$ the work of Algorithms 27, 28, 8 and 7, respectively, on an input of size n. Similarly, we denote by $S_1(n)$, $S_2(n)$, $S_3(n)$ and $S_4(n)$ the span of Algorithms 27, 28, 8 and 7, respectively, on an input of size n. Finally, we denote by N_2 and N_3 the thresholds $\mathsf{MIN_BASE}$ and $\mathsf{MIN_MERGE_BASE}$, respectively.

Since Algorithm 7 is sequential, under our worst case assumption, we clearly have $W_4(n) = S_4(n) = \Theta(n^2)$. Similarly, we have $W_1(n) = S_1(n) = \Theta(n^2)$.

Observe that, under our worst case assumption, the cardinalities of the input sets B, C differ at most by 1, when each of Algorithms 8 and 7 is called. Hence, the work of Algorithm 8 satisfies:

$$W_3(n) = \begin{cases} W_4(n) & \text{if } n \leq N_3 \\ 4W_3(n/2) & \text{otherwise.} \end{cases}$$

This implies: $W_3(n) \leq 4^{\log_2(n/N_3)} N_3^2$ for all n. Thus we have $W_3(n) = O(n^2)$. On the other hand, our assumption implies that every element of B needs to be compared with every element of C. Therefore $W_3(n) = \Theta(n^2)$ holds. Now, the span satisfies:

$$S_3(n) = \begin{cases} S_4(n) & \text{if } n \leq N_3 \\ 2S_3(n/2) & \text{otherwise.} \end{cases}$$

This implies: $S_3(n) \leq 2^{\log_2(n/N_3)} N_3^2$ for all n. Thus we have $S_3(n) = O(nN_3)$. Moreover, $S_3(n) = \Theta(n)$ holds for $N_3 = 1$.

Next, the work of Algorithm 28 satisfies:

$$W_2(n) = \begin{cases} W_1(n) & \text{if } n \leq N_2 \\ 2W_2(n/2) + W_3(n) & \text{otherwise.} \end{cases}$$

This implies: $W_2(n) \leq 2^{\log_2(n/N_2)} N_2^2 + \Theta(n^2)$ for all n. Thus we have $W_2(n) = O(nN_2) + \Theta(n^2)$.

Finally, the span of Algorithm 28 satisfies:

$$S_2(n) = \begin{cases} S_1(n) & \text{if } n \leq N_2 \\ S_2(n/2) + S_3(n) & \text{otherwise.} \end{cases}$$

Thus we have $S_2(n) = O(N_2^2 + nN_3)$. Moreover, for $N_3 = N_2 = 1$, we have $S_2(n) = \Theta(n)$.

We proceed now with cache complexity analysis, using the ideal cache model of [23]. We consider a cache of Z words where each cache line has L words. For simplicity, we assume that the elements of a given poset are packed in an array, occupying consecutive slots, each of size 1 word. We focus on the C elision of Algorithms 28 and 8, denoting by $Q_2(n)$ and $Q_3(n)$ the number of cache misses that they incur respectively on an input data of size n. We assume that the thresholds in Algorithms 28 and 8 are set to 1. Indeed, Algorithms 27 and 7 are not cache-efficient. Both may incur $\Theta(n^2/L)$ cache misses, for n large enough, whereas $Q_2(n) \in O(n/L + n^2/(ZL))$ and $Q_3(n) \in O(n^2/(ZL))$ hold, as we shall prove now. Observe first that there exist positive constants α_2 and α_3 such that we have:

$$Q_2(n) = \begin{cases} \Theta(n/L + 1) & \text{if } n \leq \alpha_2 Z \\ 2Q_2(n/2) + Q_3(n) + \Theta(1) & \text{otherwise,} \end{cases}$$

and:

$$Q_3(n) = \begin{cases} \Theta(n/L+1) & \text{if} \quad n \le \alpha_3 Z \\ 4Q_3(n/2) + \Theta(1) & \text{otherwise.} \end{cases}$$

This implies: $Q_3(n) \le 4^{\log_2(n/(\alpha_3 Z))}\Theta(Z/L)$ for all n, since $Z \in \Omega(L^2)$ holds. Thus we have $Q_3(n) \in O(n^2/(ZL))$. We deduce:

$$Q_2(n) \le 2^k\Theta(Z/L) + \sum_{i=0}^{i=k-1} 2^i Q_3(n/2^i) + \Theta(2^k)$$

where $k = \log_2(n/(\alpha_2 Z))$. This leads to: $Q_2(n) \le O(n/L + n^2/(ZL))$. Therefore, we have proved the following result.

Proposition 7. *Assume that $\mathcal{X}$ has $n \ge 1$ elements, such that neither $x \preceq y$ nor $y \preceq x$ holds for all $x, y \in \mathcal{X}$. Set the thresholds in Algorithms 28 and 8 to* 1*. Then, the work and span of* $\mathsf{Min}(\mathcal{X})$*, as computed by Algorithm 28, are* $\Theta(n^2)$ *and* $\Theta(n)$ *respectively. The cache complexity of its C elision are* $O(n/L + n^2/(ZL))$*.*

We turn now our attention to experimentation. We have implemented the operation ParallelMinPoset of Algorithm 28 as a template function in Cilk++. It is designed to work for any poset providing a method Compare(a_i, a_j) that, for any two elements a_i and a_j, determines whether $a_j \preceq a_i$, or $a_i \preceq a_j$, or a_i and a_j are incomparable. Our code offers two data structures for encoding the subsets of the poset $\mathcal{X}$: one is based on arrays and the other uses the *bag structure* introduced by the first Author in [44].

For the benchmarks reported in this section, $\mathcal{X}$ is a finite set of natural numbers compared for the divisibility relation. For example, the set of the minimal elements of $\mathcal{X} = \{6, 2, 7, 3, 5, 8\}$ is $\{2, 7, 3, 5\}$. Clearly, we implement natural numbers using the type `int` of C/C++. Since checking integer divisibility is cheap, we expect that these benchmarks could illustrate the intrinsic parallel efficiency of our algorithm.

We have benchmarked our program on sets of random natural numbers, with sizes ranging from $50,000$ to $500,000$ on a 32-core machine. This machine has 8 Quad Core AMD Opteron 8354 @ 2.2 GHz connected by 8 sockets. Each core has 64 KB L1 data cache and 512 KB L2 cache. Every four cores share 2 MB of L3 cache. The total memory is 128.0 GB. We have compared the timings with MIN_BASE and MIN_MERGE_BASE being $8, 16, 32, 64$ and 128 for different sizes of input. As a result, we choose 64 for both MIN_BASE and MIN_MERGE_BASE to reach the best timing for all the test cases.

Figure 9.1 shows the results measured by the Cilkview [33] scalability analyzer for computing the minimal elements of $100,000$ and $500,000$ random natural numbers. The reference sequential algorithm for the speedup is Algorithm 28 running on 1 core;

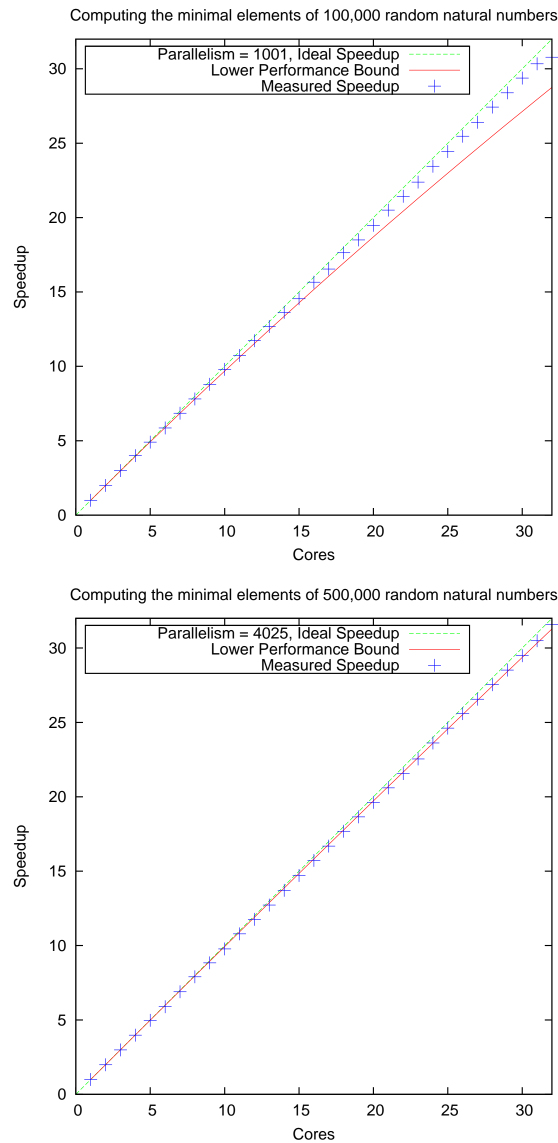


Figure 9.1: Scalability analysis for ParallelMinPoset by Cilkview

the running time of this latter code differs only by 0.5% or 1% from the C elision of Algorithm 28. On 1 core, the timing for computing the minimal elements of $100,000$ and $500,000$ random natural numbers is respectively 260 and 6454 seconds, which is slightly better (0.5%) than Algorithm 27. The number of minimal elements for the two sets of random natural numbers is respectively $99,919$ and $498,589$. These results demonstrate the abundant parallelism created by our divide-and-conquer algorithm and the very low parallel overhead of our program in Cilk++. We have also used Cilkview to check that our program is indeed free of data races.

9.3 Base Monomial Set Computation

Computation of the base monomial set as the set of all minimal elements in the pairwise Gcd set where the partial order is defined as divisibility of monomials is an inspiring application of algorithms proposed in the last section. As the Algorithms 9, 10, 11 and 12, have been presented in Chapter 6, we report in this section only the performance of them.

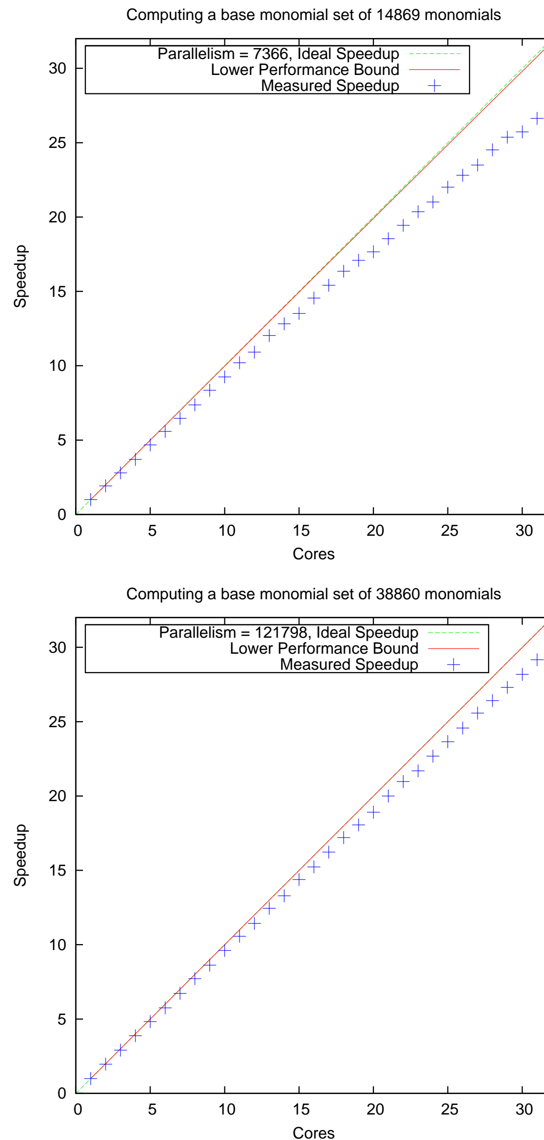


Figure 9.2: Scalability analysis for ParallelBaseMonomials by Cilkview

Figure 9.2 gives the scalability analysis results by Cilkview for computing the base monomial sets of two large monomial sets. The first one has 14869 monomials with 28 variables; its number of minimal elements here 14. Both thresholds MIN_BASE and

$\mathsf{MIN_MERGE_BASE}$ are set to 64. Its timing on 1 core is about 3.5 times less than the serial loop method, which is the function $\mathsf{SerialInnerBaseMonomials}$. Using 32 cores we gain a speedup factor of 27 with respect to the timing on 1 core. Another monomial set has 38860 monomials with 30 variables. There are 15 minimal elements. The serial loop method for this case aborted due to memory allocation failure. However, our parallel execution reaches a speedup of 30 on 32 cores. We also notice that the ideal parallelism and the lower performance bound estimated by Cilkview for both benchmarks are very high but our measured speedup curve is lower than the lower performance bound. We attribute this performance degradation to the cost of our dynamic memory allocation.

9.4 Transversal Hypergraph Generation

Hypergraphs generalize graphs in the following way. A *hypergraph* $\mathcal{H}$ is a pair $(V, \mathcal{E})$ where V is a finite set and $\mathcal{E}$ is a set of subsets of V, called the *edges* (or *hyperedges*) of $\mathcal{H}$. The elements of V are called the *vertices* of $\mathcal{H}$. The number of vertices and edges of $\mathcal{H}$ are denoted here by $n(\mathcal{H})$ and $|\mathcal{H}|$ respectively; they are called the *order* and the size of $\mathcal{H}$. We denote by $\mathsf{Min}(\mathcal{H})$ the hypergraph whose vertex set is V and whose hyperedges are the minimal elements of the poset $(\mathcal{E}, \subseteq)$. The hypergraph $\mathcal{H}$ is said *simple* if none of its hyperedges is contained in another, that is, whenever $\mathsf{Min}(\mathcal{H}) = \mathcal{H}$ holds.

We denote by $\mathsf{Tr}(\mathcal{H})$ the hypergraph whose vertex set is V and whose hyperedges are the minimal elements of the poset $(\mathcal{T}, \subseteq)$ where $\mathcal{T}$ consists of all subsets A of V such that $A \cap E \neq \emptyset$ holds for all $E \in \mathcal{E}$. We call $\mathsf{Tr}(\mathcal{H})$ the *transversal* of $\mathcal{H}$. Let $\mathcal{H}' = (V, \mathcal{E}')$ and $\mathcal{H}'' = (V, \mathcal{E}'')$ be two hypergraphs. We denote by $\mathcal{H}' \cup \mathcal{H}''$ the hypergraph whose vertex set is V and whose hyperedge set is $\mathcal{E} \cup \mathcal{E}'$. Finally, we denote by $\mathcal{H}' \vee \mathcal{H}''$ the hypergraph whose vertex set is V and whose hyperedges are the $E' \cup E''$ for all $(E', E'') \in \mathcal{E}' \times \mathcal{E}''$. The following proposition [4] is the basis of most algorithms for computing the transversal of a hypergraph.

Proposition 8. *For two hypergraphs $\mathcal{H}' = (V, \mathcal{E}')$ and $\mathcal{H}'' = (V, \mathcal{E}'')$ we have*

$$\mathsf{Tr}(\mathcal{H}' \cup \mathcal{H}'') = \mathsf{Min}(\mathsf{Tr}(\mathcal{H}') \vee \mathsf{Tr}(\mathcal{H}'')).$$

All popular algorithms for computing transversal hypergraphs, see [31, 39, 2, 19, 40], make use of the formula in Proposition 8 in an incremental manner. That is, writing $\mathcal{E} = E_1, \ldots, E_m$ and $\mathcal{H}_i = (V, \{E_1, \ldots, E_i\})$ for $i = 1 \cdots m$, these algorithms

compute $\mathsf{Tr}(\mathcal{H}_{i+1})$ from $\mathsf{Tr}(\mathcal{H}_i)$ as follows

$$\mathsf{Tr}(\mathcal{H}_{i+1}) = \mathsf{Min}(\mathsf{Tr}(\mathcal{H}_i) \vee (V, \{\{v\} \ \mid \ v \in E_{i+1}\}))$$

Input : A hypergraph $\mathcal{H}$
Output : $\mathsf{Tr}(\mathcal{H})$

```
1 if |H| ≤ TR_BASE then
2   return SerialTransversal(H);
3 (H⁻, H⁺) ← Split(H);
4 H⁻ ← spawn ParallelTransversal(H⁻);
5 H⁺ ← spawn ParallelTransversal(H⁺);
6 sync;
7 return ParallelHypMerge(H⁻, H⁺);
```

Algorithm 29: ParallelTransversal

The differences between these algorithms consist of various techniques to minimize the construction of unnecessary intermediate hyperedges. While we believe that these techniques are all important, we propose to apply Berge's formula *à la lettre*, that is, to divide the input hypergraph $\mathcal{H}$ into hypergraphs $\mathcal{H}'$, $\mathcal{H}''$ of similar sizes and such that $\mathcal{H}' \cup \mathcal{H}'' = \mathcal{H}$. Our intention is to create opportunity for parallel execution. At the same time, we want to control the intermediate expression swell resulting from the computation of

$$\mathsf{Tr}(\mathcal{H}) \vee \mathsf{Tr}(\mathcal{H}').$$

To this end, we compute this expression in a divide-and-conquer manner and apply the Min operator to the intermediate results.

Algorithm 29 is our main procedure. Similarly to Algorithm 28, it proceeds in a divide-and-conquer manner with a threshold. For the base case, we call $\mathsf{SerialTransversal}(\mathcal{H})$, which can implement any serial algorithms for computing the transversal of hypergraph $\mathcal{H}$. When the input hypergraph is large enough, then this hypergraph is split into two so as to apply Proposition 8 with the two recursive calls performed concurrently. When these recursive calls return, their results are merged by means of Algorithm 30.

Given two hypergraphs $\mathcal{H}$ and $\mathcal{K}$, with the same vertex set, satisfying $\mathsf{Tr}(\mathcal{H}) = \mathcal{H}$ and $\mathsf{Tr}(\mathcal{K}) = \mathcal{K}$, the operation ParallelHypMerge of Algorithm 30 returns $\mathsf{Min}(\mathcal{H} \vee \mathcal{K})$. This operation is another instance of an application where the poset can be so large that it is desirable to compute its minimal elements concurrently to the generation

```
Input   : H, K such that Tr(H) = H and Tr(K) = K.
Output  : Min(H ∨ K)
1  if |H| ≤ MERGE_HYP_BASE and
2     |K| ≤ MERGE_HYP_BASE then
3     return SerialHypMerge(H, K);
4  else if |H| > MERGE_HYP_BASE and
5        |K| > MERGE_HYP_BASE then
6     (H⁻, H⁺) ← Split(H);
7     (K⁻, K⁺) ← Split(K);
8     L ← spawn
9           HalfParallelHypMerge(H⁻, K⁻, H⁺, K⁺);
10    M ← spawn
11          HalfParallelHypMerge(H⁻, K⁺, H⁺, K⁻);
12    return Union(ParallelMinMerge(L, M));
13 else if |H| > MERGE_HYP_BASE and
14       |K| ≤ MERGE_HYP_BASE then
15    (H⁻, H⁺) ← Split(H);
16    M⁻ ← ParallelHypMerge(H⁻, K);
17    M⁺ ← ParallelHypMerge(H⁺, K);
18    return Union(ParallelMinMerge(M⁻, M⁺));
19 else
      // |H| ≤ MERGE_HYP_BASE and
      // |K| > MERGE_HYP_BASE
20    (K⁻, K⁺) ← Split(K);
21    M⁻ ← ParallelHypMerge(K⁻, H);
22    M⁺ ← ParallelHypMerge(K⁺, H);
23    return Union(ParallelMinMerge(M⁻, M⁺));
```

Algorithm 30: ParallelHypMerge

```
Input   : four hypergraphs H, K, L, M
Output  : Min(Min(H ∨ K) ∪ Min(L ∨ M))
1 N ← spawn ParallelHypMerge(K, H);
2 P ← spawn ParallelHypMerge(L, M);
3 sync;
4 return Union(ParallelMinMerge(N, P));
```

Algorithm 31: HalfParallelHypMerge

of the poset itself, thus avoiding storing the entire poset in memory. As for the application described in Section 9.3, one can indeed efficiently generate the elements of the poset and compute its minimal elements simultaneously.

The principle of Algorithm 30 is very similar to that of Algorithm 8. Thus, we should simply mention two points. First, Algorithm 30 uses a subroutine, namely HalfParallelHypMerge of Algorithm 31, for clarity. Secondly, the base case of Algorithm 30, calls $\mathsf{SerialHypMerge}(\mathcal{H}, \mathcal{K})$, which can implement any serial algorithms for computing $\mathsf{Min}(\mathcal{H} \vee \mathcal{K})$.

We have implemented our algorithms in Cilk++ and benchmarked our code with some well-known problems on the same 32-core machine reported in Section 9.2. An implementation detail which is worth to mention is data representation. We represent each hyperedge as a bit-vector. For a hypergraph with n vertices, each hyperedge is encoded by n bits. By means of this representation, the operations on the hyperedges such as inclusion test and union can be reduced to bit operations. Thus, a hypergraph with m edges is encoded by an array of $m\,n$ bits. Traversing the hyperedges is simply by moving pointers to the bit-vectors in this array.

Our test problems can be classified into three groups. The first one consists of three types of large examples reported in [39]. We summarize their features and compare the timing results in Table 9.1. A scalability analysis for the three large problems in data mining on a 32-core is illustrated in Figure 9.3. The second group considers an enumeration problem (Kuratowski hypergraph), as listed in Table 9.2 and Figure 9.4. The third group is Lovasz hypergraph [4], reported in Table 9.3. The sizes of the three base cases used here (TR_BASE, MERGE_HYP_BASE and MIN_MERGE_BASE) are respectively 32, 16 and 128. Our experimentation shows that the base case threshold is an important influential factor on performance. In this work, they are determined by our test runs. To predict the feasible values based on the type of a poset and the hierarchical memory of a machine would definitely help. We shall develop a tool for this purpose when we deploy our software.

In Table 9.1, we describe the parameters of each problem following the same notation as in [39]. The first three columns indicate respectively the number of vertices, n, the number of hyperedges, m, and the number of minimal transversals, t. The problems classified as *Threshold*, *Dual Matching* and *Data Mining* are large examples selected from [39]. We have used `thg`, a Linux executable program developed by Kavvadias and Stavropoulos in [39] for their algorithm, named KS, to measure the time for solving these problems on our machine. We observed that the timing results of `thg` on our machine were very close to those reported in [39]. Thus, we show

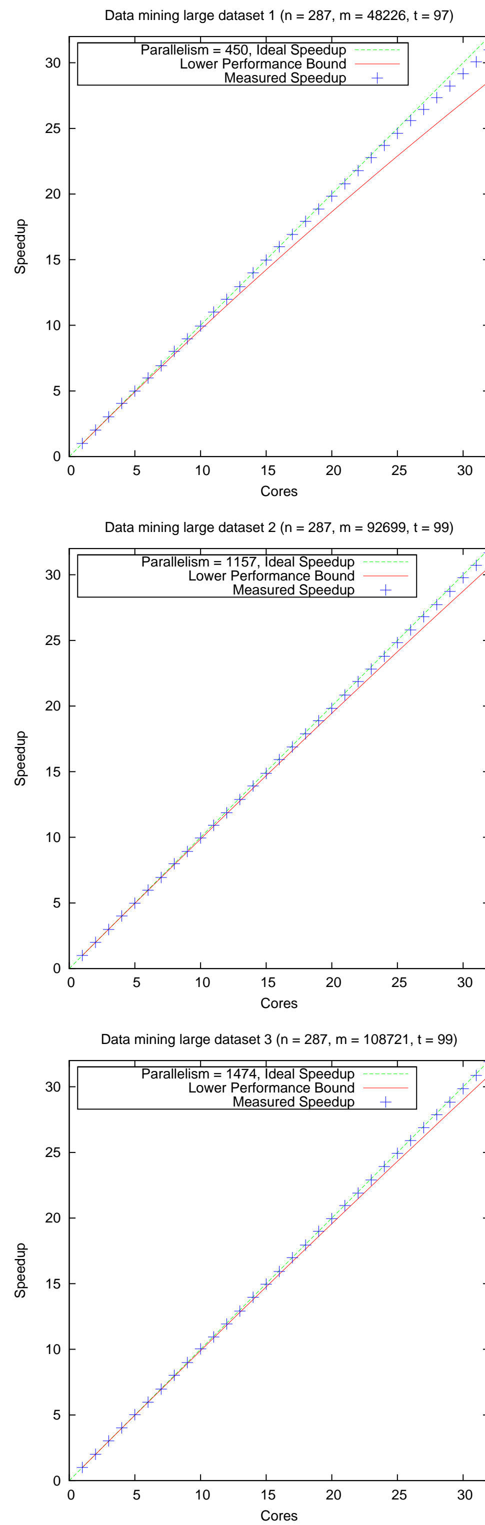


Figure 9.3: Scalability analysis on ParallelTransversal for data mining problems by Cilkview

here the timing results (seconds) presented in [39] in the fourth column (KS) in our Table 9.1. From the comparisons in [39], the KS algorithm outperforms the algorithm of Bailey et al. given in [2] (BMR) and the algorithm of Fredman and Khachiyan as implemented by Boros et al. in [6] (BEGK) for the *Dual Matching* and *Threshold* graphs. However, for the three large problems from data mining, the KS algorithm is about 30 to 60 percent slower than the best ones between BEGK and BMR.

In the last three columns in Table 9.1, we report the timing (in seconds) of our program for solving these problems using 1 core and 32 cores, and the speedup factor on 32-core w.r.t on 1-core. On 1-core, our method is about 6 to 18 times faster for the selected *Dual Matching* problems and the large problems in data mining. Our program is particularly efficient for the *Threshold* graphs, for which it takes only about 0.01 seconds for each of them, while `thg` took about 11 to 82 seconds. In addition, our method shows significant speedup on multi-cores for the problems of large input size. As shown in Figure 9.3, for the three data mining problems, our code demonstrates linear speedup on 32 cores w.r.t the timing of the same algorithm on 1 core.

There are three sets of hypergraphs in [39] on which our method does not perform well, namely *Matching*, *Self-Dual Threshold* and *Self-Dual Fano-Plane* graphs. For these examples our code is about 2 to 50 times slower than the KS algorithm presented in [39]. Although the timing of such examples is quite small (from 0.01 to 178 s), they demonstrate the efficient techniques used in [39]. In-cooperating such techniques into our algorithm is our future work.

Instance parameters			KS	ParallelTransversal		Speedup Ratio	
n	m	t	(s)	1-core (s)	32-core (s)	KS/1-core	KS/32-core
Threshold problems							
140	4900	71	11	0.01	-	1000	-
160	6400	81	23	0.01	-	2000	-
180	8100	91	44	0.01	-	4000	-
200	10000	101	82	0.02	-	4000	-
Dual Matching problems							
34	131072	17	57	9	0.57	6	100
36	262144	18	197	23	1.77	9	111
38	524288	19	655	56	3.53	12	186
40	1048576	20	2167	131	7.13	17	304
Data Mining problems							
287	48226	97	1648	92	3	18	549
287	92699	99	6672	651	21	10	318
287	108721	99	9331	1146	36	8	259

Table 9.1: Tests for examples from [39]

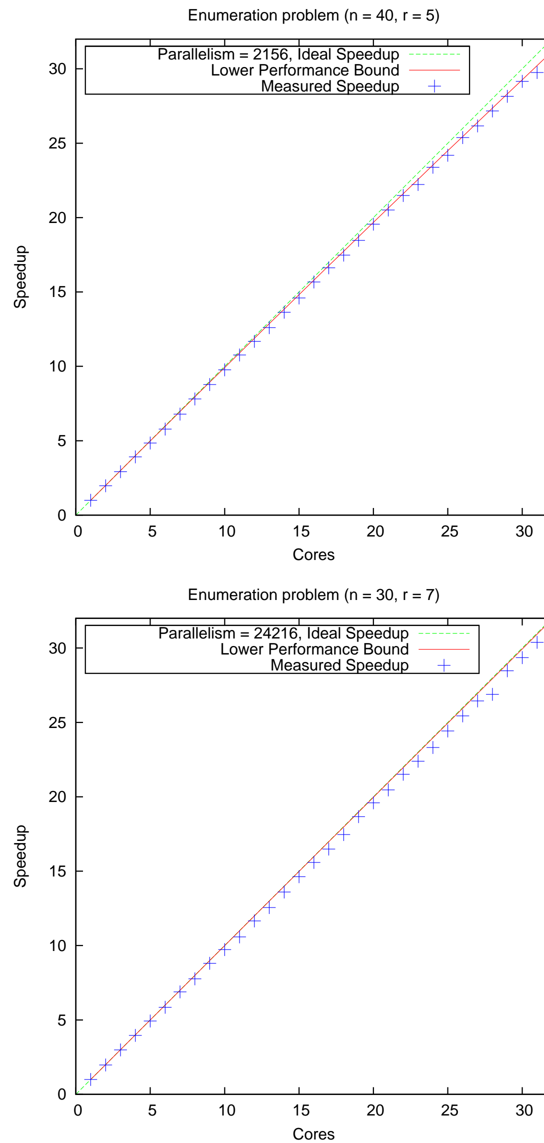


Figure 9.4: Scalability analysis on ParallelTransversal for K_{40}^5 and K_{30}^7 by Cilkview

The first family of classical hypergraphs that we have tested is related to an enumeration problem, namely the Kuratowski K_n^r hypergraphs. Table 9.2 gives two representative ones. This type of hypergraphs are defined by two parameters n and r. Given n distinct vertices, such a hypergraph contains all the hyperedges that have exactly r vertices. Our program achieves linear speedup on this class of hypergraphs with sufficiently large size, as reported in Table 9.2 and Figure 9.4 for K_{40}^5 and K_{30}^7. We have also used the `thg` program provided by the Authors of [39] to solve these problems. The timing for solving K_{30}^5 by the `thg` program is about 6500 seconds, which is about 70 times slower than our ParallelTransversal on 1-core. For the case of

K^5_{40} and K^7_{30}, the `thg` program did not produce a result after running for more than 15 hours.

Instance parameters				KS	ParallelTransversal				
n	r	m	t	(s)	1-core	16-core		32-core	
					(s)	(s)	Speedup	(s)	Speedup
30	5	142506	27405	6500	88	6	14.7	3.5	25.0
40	5	658008	91390	>15 hr	915	58	15.8	30	30.5
30	7	2035800	593775	>15 hr	72465	4648	15.6	2320	31.2

Table 9.2: Tests for the Kuratowski hypergraphs

Another classical hypergraph is the Lovasz hypergraph, which is defined by a positive integer r. Consider r finite disjoint sets $X_1, \ldots, X_r$ such that X_j has exactly j elements, for $j = 1 \cdots r$. The Lovasz hypergraph of rank r, denoted by L_r, has all its hyperedges of the form

$$X_j \cup \{x_{j+1}, \ldots, x_r\},$$

where $x_{j+1}, \ldots, x_r$ belong respectively to $X_{j+1}, \ldots, X_r$, for $j = 1 \ldots r$. We have tested our implementation with the Lovasz hypergraphs up to rank 10. For the rank 9 problem, we obtained 25 speedup on 32-core. For the one of rank 10, due to time limit, we only obtained the timing on 32-core and 16-core, which shows a linear speedup from 16 cores to 32 cores. The `thg` program solves the problem of rank 8 in 8000 seconds. For the problems of rank 9 and 10, the `thg` program did not complete within 15 hours.

Instance parameters				KS	ParallelTransversal				
n	r	m	t	(s)	1-core	16-core		32-core	
					(s)	(s)	Speedup	(s)	Speedup
36	8	69281	69281	8000	119	13	8.9	10	11.5
45	9	623530	623530	>15 hr	8765	609	14.2	347	25.3
55	10	6235301	6235301	>15 hr	-	60509	-	30596	

Table 9.3: Tests for the Lovasz hypergraphs

9.5 Some Remarks

The implementation in Cilk++ on multi-cores of the proposed a parallel algorithm for computing the minimal elements of a finite poset is capable of processing large posets that a serial implementation could not process. Moreover, for sufficiently large input data set, our code reaches linear speedup on 32 cores.

We control intermediate expression swell by generating the poset and computing its minimal elements concurrently. Our Cilk++ code for computing transversal hypergraphs is competitive with the implementation reported by Kavvadias and Stavropoulos in [39]. Moreover, our code outperforms the one of our colleagues on three sets of large input problems, in particular the problems from data mining. However, our code is slower than theirs on other data sets. In fact, our code is a preliminary implementation, which simply applies Berge's formula in a divide-and-conquer manner. We still need to enhance our implementation with the various techniques which have been developed for controlling expression swell in transversal hypergraph computations [2, 19, 31, 39, 40].

This work can be extended in different directions. For example, a deeper complexity analysis of our algorithm for computing the minimal elements of a finite poset could be discussed. It is also interesting to adapt this algorithm to the computation of GCD-free bases and the removal of redundant components.

Chapter 10

Conclusions and Future Work

Minimizing the evaluation cost of a polynomial expression is a fundamental problem in computer science. In this thesis, we have proposed tools that, for a polynomial f given as the sum of its terms, compute a *syntactic decomposition* of it that permits a more efficient evaluation. Our algorithm runs in $O(t^9 nd \log t)$ bit operations plus $O(t^9 d)$ operations in the base field where d, n and t are the total degree, number of variables and number of terms of f. These estimation is based on the worst case where the cardinality of base monomial set is quadratic with the number of terms in the input polynomial. In practice, given a large dense polynomial, its base monomial set is often the variable set which is much smaller than our estimation. We have implemented our algorithm in the `Cilk++` concurrency platform and our implementation achieves near linear speedup on 16 cores with large enough input. Besides, we have introduced an algorithm to generate a parallel evaluation schedule of a syntactic decomposition. Our experimental results reported in Chapter 8 illustrate that our approach can handle much larger polynomials than other available software solutions. Moreover, for some large polynomial, our resulting schedule provides a 10-times-faster evaluation comparing with the direct evaluation of its straight-line program (SLP) representation (both conducted sequentially).

We indicate two possible relaxation of our hypergraph method. As mentioned in Chapter 3, our definition on syntactic operation is designed to avoid factorizations of the following typical form

$$f = x^n - 1 = (x - 1)(x^n + x^{n-1} + \cdots + 1),$$

where expanding the product leads to term cancellation. Our syntactic decomposition imposes the condition that after expanding it there is no combination of monomials,

such that its evaluation cost never grows comparing to the input expanded polynomial. However, distributing a term may sometimes lead to a better representation than syntactic decomposition. Consider this polynomial in $Z[x, y, a, b, c, d]$:

$$f = xy + xd + yb + ac + ad + bc + 2bd.$$

Its syntactic decomposition computed by our hypergraph method is

$$x(y + d) + a(c + d) + (y + c + 2d)b$$

which requires 6 additions and 4 multiplications to evaluate whereas there exists a better representation of f

$$(a + b)(c + d) + (x + b)(y + d)$$

which allows evaluation of f with 5 additions and 2 multiplications.

To this end, we discuss two possible relaxations on the algorithm ParSynFactorization. The first one aims at computing factorization of the form $(a + b)(a + b)$. Consider algorithm ParSynFactorization with input $f = a^2 + 2ab + b^2$. Our definition of syntactic factorization and thus the algorithm ParSynFactorization prevents the factorization $f = (a + b)(a + b)$ due to the combination of monomial ab. When we get the monomial set $M = \{a, b\}$ of the base and monomial set $Q = \{a, b\}$ of cofactor in a candidate syntactic factorization, we compute their product monomial set N as in Line 18 of Algorithm 3. We find that $N = \{a^2, ab, b^2\}$ thus $|N| \neq |M| \cdot |Q|$. We are then led to Line 25 and a monomial in the monomial set $Q = \{a, b\}$, say b, will be randomly removed. The algorithm terminates with the partial syntactic factorization

$$f = a\,(a + 2\,b) + b^2.$$

Evaluating it requires two more multiplications than evaluating

$$f = (a + b)(a + b).$$

Thus we propose the first relaxation that we do not immediately break the computation of a syntactic factorization (directly goes to Line 25 once $|N| \neq |M| \cdot |Q|$) but continue attempting to build a syntactic factorization. We continue to solve

coefficients of the candidate syntactic factorization by the following equality,

$$(g_1 a + g_2 b)(h_1 a + h_2 b) = a^2 + 2ab + b^2$$

$$\begin{cases} g_1 h_1 &= 1 \\ g_1 h_2 + g_2 h_1 &= 2 \\ g_2 h_2 &= 1 \end{cases} \xRightarrow{g_1=1} \begin{cases} g_1 &= 1 \\ g_2 &= 1 \\ h_1 &= 1 \\ h_2 &= 1 \end{cases}$$

which leads to the factorization $f = (a+b)(a+b)$. Note that to solve coefficients of this factorization, we are lead to a non-linear equation system.

Once we allow combination of monomials in algorithm ParSynFactorization, the next possible relaxation is on the computation of coefficients of a candidate syntactic factorization. Note that the system to solve the coefficients contains $|M|\cdot|Q|$ equations and $|M|+|Q|$ unknowns. Hence it is possible that the system does not have solutions. As described in Proposition 6, we first set g_1 to one, and then the value of $h_1, \ldots, h_b$ and then of $g_2, \ldots, g_a$ can be deduced using $a+b-1$ equations. After that, we use the remaining equations to verify these values. The second relaxation we proposed is that if these values do not lead to a solution, instead of rejecting these values, we allow the necessary distribution of terms in the input polynomial. For example, for the input $f = ac + ad + bc + 2bd$, after obtaining the two monomial sets $M = \{a, b\}$, $Q = \{c, d\}$ and the product monomial set $N = \{ac, ad, bc, bd\}$, we set up the following system to solve the candidate coefficients.

$$(g_1 a + g_2 b)(h_1 c + h_2 d) = ac + ad + bc + 2bd$$

$$\begin{cases} g_1 h_1 &= 1 \\ g_1 h_2 &= 1 \\ g_2 h_1 &= 1 \end{cases} \xRightarrow{g_1=1} \begin{cases} h_1 &= 1 \\ h_2 &= 1 \\ g_2 &= 1 \end{cases} \Rightarrow \left\{ g_2 h_2 \neq 2 \right.$$

Though there does not exist solutions to this system, we do not reject this candidate partial factorization but rewrite the polynomial f as $f = ac + ad + bc + bd + bd$. Then the partial factorization $f = (a+b)(c+d) + bd$ can be produced.

If these relaxations are applied, the optimized expression may not be a syntactic decomposition but it provides opportunity to produce a more efficient representation. We point out the above two possible relaxation on our optimization algorithm as future work.

Appendix A

An Example Illustrating the Typical Output of Different Optimization Techniques for a Polynomial Expression

Input polynomial (557 operations):

$$a^2b^2cde^2ghjlm + a^2b^2cdeh^2klmn + a^2bcdegh^2lm^2n + ab^3c^2de^2ghij+$$
$$ab^3c^2deh^2ikn + ab^2c^2de^2ghjlm + ab^2c^2degh^2imn + ab^2c^2deh^2klmn+$$
$$ab^2cd^2e^3ghjm + ab^2cd^2e^2h^2kmn + abc^2degh^2lm^2n + abcd^2e^2gh^2m^2n+$$
$$b^3c^3de^2ghij + b^3c^3deh^2ikn + b^2c^3degh^2imn + b^2c^2d^2e^3ghjm+$$
$$b^2c^2d^2e^2h^2kmn + bc^2d^2e^2gh^2m^2n + ab^2ceghij^2lm + ab^2ch^2ijklmn+$$
$$abc^2de^2ghjlm + abc^2deh^2klmn + abcgh^2ijlm^2n + abe^2ghjklm^2n+$$
$$abeh^2k^2lm^2n^2 + ac^2degh^2lm^2n + aegh^2klm^3n^2 + b^3c^2eghi^2j^2+$$
$$b^3c^2h^2i^2jkn + b^2c^3de^2ghij + b^2c^3deh^2ikn + b^2c^2gh^2i^2jmn+$$
$$b^2cde^2ghij^2m + b^2cdeh^2ijkmn + b^2ce^2ghijkmn + b^2ceh^2ik^2mn^2+$$
$$bc^3degh^2imn + bc^2d^2e^3ghjm + bc^2d^2e^2h^2kmn + bcdegh^2ijm^2n+$$
$$bcegh^2ikm^2n^2 + bde^3ghjkm^2n + bde^2h^2k^2m^2n^2 + c^2d^2e^2gh^2m^2n+$$
$$de^2gh^2km^3n^2$$

Maple's Horner scheme (382 operations):

$$de^2gh^2km^3n^2 + c^2d^2e^2gh^2m^2n + ((k^2m^2n^2h^2 + jkm^2nghe)e^2d+ (egh^2ikm^2n^2 + degh^2ijm^2n + ((h^2m^2ng + mh^2kn + jmghe)e^2d^2+ dih^2nmegc)c)c + (((k^2n^2imh^2 + jiknmghe)e + (ijmh^2kn + ij^2mghe)ed+ (i^2jgh^2mn + (mh^2kn + jmghe)e^2d^2 + (ih^2nmg + ih^2kn + ijghe)edc)c)c+ (i^2j^2egh + i^2jh^2kn + (ih^2kn + ijghe)edc)c^2b)b)b + (c^2degh^2lm^2n + egh^2klm^3n^2+ ((lk^2m^2n^2h^2 + jkm^2nlghe)e + (h^2m^2nd^2e^2g + lm^2ijgh^2n + (h^2m^2nlg + lmh^2kn+ jlmghe)edc)c + ((ij^2lmegh + jlimh^2kn + (mh^2kn + jmghe)e^2d^2 + (ih^2nmg+ lmh^2kn + jlmghe)edc)c + (ih^2kn + ijghe)edc^2b)b)b + (h^2m^2ndcleg + (lmh^2kn+ jlmghe)edcb)ba)a$$

Maple's optimize with 'tryhard' option (153 operations):

$$t_{86} = al, t_{85} = de, t_{84} = ij, t_{83} = kn, t_{55} = c^2, t_{82} = at_{55}, t_{46} = m^2, t_{81} = bt_{46},$$
$$t_{57} = b^2, t_{80} = ct_{57}, t_{44} = n^2, t_{79} = kt_{44}, t_{78} = t_{57}m, t_{77} = a^2l, t_{76} = t_{57} + b,$$
$$t_{52} = e^2, t_{53} = d^2, t_{75} = t_{53}t_{52}, t_{56} = bt_{57}, t_{74} = t_{55}t_{56}, t_{73} = et_{86}, t_{72} = it_{78},$$
$$t_{71} = t_{55}t_{75}, t_{49} = i^2, t_{70} = t_{49}t_{74}, t_{69} = t_{81}t_{83}, t_{68} = ct_{72}, t_{67} = it_{52}t_{80}, t_{54} = ct_{55},$$
$$t_{66} = (t_{56} + t_{57})t_{54}, t_{65} = t_{84} + t_{77}, t_{64} = (t_{75} + lt_{84})a, t_{63} = (dt_{73} + t_{75})t_{55},$$
$$t_{51} = et_{52}, t_{50} = h^2, t_{48} = j^2, t_{45} = mt_{46},$$
$$t1 = ((et_{68} + (t_{73} + dt_{52})t_{81})k^2t_{44} + (jt_{70} + (at_{74} + t_{66})it_{85} + (bt_{63} + (t_{63}+ (t_{65}t_{85} + t_{64})c)t_{57})m)t_{83})t_{50} + (((t_{45}t_{86} + cit_{81})et_{79} + (t_{55}t_{49}jt_{78} + (t_{71}+ (t_{71} + ct_{64})b)t_{46})n)t_{50} + ((t_{68}t_{86} + t_{70})t_{48}e + (t_{52}t_{69}t_{86} + (t_{67}t_{83} + (at_{80}+ t_{76}t_{55})t_{53}t_{51})m)j)h + ((t_{52}t_{45}t_{79} + (t_{54}t_{72} + (lt_{46} + t_{72})t_{82} + (t_{54}im + (lt_{82}+ t_{65}c)t_{46})b)ne)t_{50} + (t_{48}mt_{67} + (t_{51}t_{69} + (ct_{77}t_{78} + it_{66} + (t_{56}i + t_{76}ml)t_{82})t_{52})j)h)d)g$$

Syntactic decomposition (142 operations):

$$(((((bej + hmn)d(ab + bc + c) + b^2ij^2)c + (bej + hmn)kmn)(al + de)+$$
$$(bej + hmn)bcikn + (c(ab + bc + c) + jm)bcdihn)m + (bij + de(ab+$$
$$bc + c))b^2c^2ij)egh + ((((((al + de)(ab + bc + c) + bij)d + knbi)e + ajlbi)c+$$
$$(al + de)kmne)m + (bij + de(ab + bc + c))bc^2i)bh^2kn + (alm + bci)bcijgh^2mn$$

Syntactic decomposition + CSE (94 operations):

$$t_5 = bej + hmn, t_8 = bc, t_9 = ab + t_8 + c, t_{11} = b^2, t_{13} = j^2, t_{18} = mn,$$
$$t_{21} = al, t_{22} = de, t_{23} = t_{21} + t_{22}, t_{41} = bi, t_{42} = t_{41}j, t_{44} = t_{42} + t_{22}t_9,$$
$$t_{46} = c^2, t_{47} = t_{46}i, t_{76} = h^2,$$
$$t_{91} = ((((t_5dt_9 + t_{11}it_{13})c + t_5kt_{18})t_{23} + t_5bcikn + (ct_9 + jm)bcdihn)m+$$
$$t_{44}t_{11}t_{47}j)egh + (((((t_{23}t_9 + t_{42})d + knt_{41})e + ajlbi)c + t_{23}kt_{18}e)m+$$
$$t_{44}bt_{47})bt_{76}kn + (t_{21}m + t_8i)bcijgt_{76}mn$$

Bibliography

[1] Jounaidi Abdeljaoued and Henri Lombardi. *Méthodes Matricielles: Introduction à la Complexité Algébrique*. Springer, 2003.

[2] James Bailey, Thomas Manoukian, and Kotagiri Ramamohanarao. A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns. In *ICDM '03: Proceedings of the Third IEEE International Conference on Data Mining*, page 485, Washington, DC, USA, 2003. IEEE Computer Society.

[3] E.G. Belaga. Evaluation of polynomials of one variable with preliminary processing of the coeffcients. *Problemy Kibernet*, 5:7–15, 1961.

[4] Claude Berge. *Hypergraphes : combinatoire des ensembles finis*. Gauthier-Villars, 1987.

[5] Stuart J. Berkowitz. On computing the determinant in small parallel time using a small number of processors. *Inf. Process. Lett.*, 18(3), 1984.

[6] Endre Boros, KLeonid hachiyan, Khaled Elbassioni, and Vladimir Gurvich. An efficient implementation of a quasi-polynomial algorithm for generating hypergraph transversals. In *Proc. of the 11th European Symposium on Algorithms (ESA)*, volume 2432, pages 556–567. LNCS, Springer, 2003.

[7] Robert K. Brayton and Curtis T. McMullen. The decomposition and factorization of boolean expressions. In *Proc. Int. Symp. Circuits Systems*, pages 49–54, 1982.

[8] Robert K. Brayton, Richard Rudell, Alberto Sangiovarmi-Vincentelli, and Albert Wang. Mis: Multiple-level logic optimization system. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 1062–1081, 1987.

[9] Robert K. Brayton, Richard Rudell, Alberto Sangiovarmi-Vincentelli, and Albert Wang. Multi-level logic optimization and the rectangular covering problem. In *Proc. Int. Conf. Compu.-Adoded Des*, pages 66–69, 1987.

[10] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[11] Melvin A. Breuer. Generation of optimal code for expressions via factorization. *Commun. ACM*, 12(6):333–340, 1969.

[12] Peter Bürgisser, Michael Clausen, and Amin Shokrollahi. *Algebraic complexity theory*. Springer-Verlag, Berlin, 1997.

[13] J. Carnicer and M. Gasca. Evaluation of multivariate polynomials and their derivatives. *Mathematics of Computation*, 54(189):231–243, 1990.

[14] Martine Ceberio and Vladik Kreinovich. Greedy algorithms for optimizing multivariate horner schemes. *SIGSAM Bull.*, 38(1):8–15, 2004.

[15] Changbo Chen, François Lemaire, Marc Moreno Maza, Wei Pan, and Yuzhen Xie. Efficient computations of irredundant triangular decompositions with the `regularchains` library. In *Proc. of the International Conference on Computational Science (2)*, volume 4488 of *Lecture Notes in Computer Science*, pages 268–271. Springer, 2007.

[16] Cilk Arts. Cilk++. http://www.cilk.com/.

[17] Robert M. Corless, David J. Jeffrey, and Michael B. Monagan. Two perturbation calculations in fluid mechanics using large-expression management. *J. Symb. Comput.*, 23(4):427–443, 1997.

[18] Matthew T. Dickerson. The functional decomposition of polynomials. Technical report, Ithaca, NY, USA, 1989.

[19] Guozhu Dong and Jinyan Li. Mining border descriptions of emerging patterns from dataset pairs. *Knowl. Inf. Syst.*, 8(2):178–202, 2005.

[20] Thomas Eiter and Georg Gottlob. Hypergraph transversal computation and related problems in logic and ai. In *JELIA '02: Proceedings of the European Conference on Logics in Artificial Intelligence*, pages 549–564, London, UK, 2002. Springer-Verlag.

[21] Jean-Charles Faugère, Joachim von zur Gathen, and Ludovic Perret. Decomposition of generic multivariate polynomials. In *ISSAC '10: Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, pages 131–137, New York, NY, USA, 2010. ACM.

[22] Timothy S. Freeman, Gregory M. Imirzian, Erich Kaltofen, and Lakshman Yagati. Dagwood: a system for manipulating polynomials given by straight-line programs. *ACM Trans. Math. Softw.*, 14(3):218–240, 1988.

[23] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, New York, October 17–19 1999.

[24] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[25] Sanchit Garg and Éric Schost. Interpolation of polynomials given by straight-line programs. *Theor. Comput. Sci.*, 410(27-29):2659–2662, 2009.

[26] Joachim von zur Gathen. Functional decomposition of polynomials: the tame case. *J. Symb. Comput.*, 9(3):281–299, 1990.

[27] Joachim von zur Gathen. Functional decomposition of polynomials: The wild case. *J. Symb. Comput.*, 10(5):437–452, 1990.

[28] M. Giusti, J. Heintz, J. E. Morais, J. Morgenstern, and L. M. Pardo. Straight-line programs in geometric elimination theory. *Journal of Pure and Applied Algebra*, 124:1–3, 1998.

[29] R. L. Graham. Bounds on multiprocessing anomalies and related packing algorithms. In *AFIPS '72 (Spring): Proceedings of the May 16-18, 1972, spring joint computer conference*, pages 205–217, New York, NY, USA, 1972. ACM.

[30] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharma. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28(2):140–174, 2003.

[31] Matthias Hagen. Lower bounds for three algorithms for transversal hypergraph generation. *Discrete Appl. Math.*, 157(7):1460–1469, 2009.

[32] U.-U. Haus, S. Klamt, and T. Stephen. Computing knock out strategies in metabolic networks. *ArXiv e-prints*, 2008.

[33] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The cilkview scalability analyzer. In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 145–156, New York, NY, USA, 2010. ACM.

[34] William George Horner. A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, pages 308–335, 1819.

[35] Anup Hosangadi, Fallah Fallah, and Ryan Kastner. Factoring and eliminating common subexpressions in polynomial expressions. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 169–174, Washington, DC, USA, 2004. IEEE Computer Society.

[36] Gabriela Jeronimo and Juan Sabia. Computing multihomogeneous resultants using straight-line programs. *J. Symb. Comput.*, 42(1-2):218–235, 2007.

[37] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[38] Erich Kaltofen. Greatest common divisors of polynomials given by straight-line programs. *J. ACM*, 35(1):231–264, 1988.

[39] Dimitris J. Kavvadias and Elias C. Stavropoulos. An efficient algorithm for the transversal hypergraph generation. *Journal of Graph Algorithms and Applications*, 9(2):239–264, 2005.

[40] Leonid Khachiyan, Endre Boros, Khaled M. Elbassioni, and Vladimir Gurvich. A new algorithm for the hypergraph transversal problem. In *COCOON*, pages 767–776, 2005.

[41] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[42] Charles E. Leiserson, Liyun Li, Marc Moreno Maza, and Yuzhen Xie. Efficient evaluation of large polynomials. In *Proc. International Congress of Mathematical Software - ICMS 2010*. Springer, 2010.

[43] Charles E. Leiserson, Liyun Li, Marc Moreno Maza, and Yuzhen Xie. Parallel computation of the minimal elements of a poset. In *Proc. PASCO'10*. ACM Press, 2010.

[44] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 303–314, New York, NY, USA, 2010. ACM.

[45] Juan Llovet, Bonifacio Castaño, and Raquel Martínez. Computing the characteristic polynomial of multivariate polynomial matrices given by straight-line programs. *Math. Comput. Simul.*, 45(1-2):39–57, 1998.

[46] Gary L. Miller, Vijaya Ramachandran, and Erich Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. Comput.*, 17(4):687–695, 1988.

[47] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *SFCS '85: Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 478–489, Washington, DC, USA, 1985. IEEE Computer Society.

[48] T.S. Motzkin. Evaluation of polynomials. *Bull. Amer. Math. Soc.*, 61:163, 1955.

[49] Bill Naylor. *Polynomial GCD Using Straight Line Program Representation*. PhD thesis, University of Bath, United Kingdom, 2000.

[50] Alexander Ostrowski. On two problems in abstract algebra connected with horner's rule. *Studies in Mathematics and Mechanics presented to Richard von Mises*, pages 40–48, 1954.

[51] Victor Pan. Methods of computing values of polynomials. *Russian Mathematical Surveys*, 21(1):105, 1966.

[52] J. M. Peña. On the multivariate horner scheme. *SIAM J. Numer. Anal.*, 37(4):1186–1197, 2000.

[53] J. M. Peña and Thomas Sauer. On the multivariate horner scheme ii: running error analysis. *Computing*, 65(4):313–322, 2000.

[54] Nathalie Revol and Jean-Louis Roch. Parallel evaluation of arithmetic circuits. *Theor. Comput. Sci.*, 162(1):133–150, 1996.

[55] Saswati Sarkar and Kumar N. Sivarajan. Hypergraph models for cellular mobile communication systems. *IEEE Transactions on Vehicular Technology*, 47(2):460–471, 1998.

[56] D Stoutemyer. Which polynomial representation is best? surprises abound! In *Proceedings of the 1984 Macsyma Users' Conference*, pages 221–243, New York, NY, USA, 1984.

[57] Leslie G. Valiant, Sven Skyum, S. Berkowitz, and Charles Rackoff. Fast parallel computation of polynomials using few processors. *SIAM J. Comput.*, 12(4):641–644, 1983.

Curriculum Vitae

Name:	**Liyun Li**
Post-Secondary Education and Degrees:	The University of Western Ontario London, Ontario, Canada M.Sc. Computer Science, August 2010 Beihang University Beijing, China B.Sc. in Information and Computational Science, July 2004
Work Experience:	Research Assistant & Teaching Assistant. University of Western Ontario, London, Canada. Sept. 2008 - Dec. 2009
Publications:	Charles E. Leiserson, Liyun Li, Marc Moreno Maza and Yuzhen Xie (2010) Efficient Evaluation of Large Polynomials. Proceedings of International Congress of Mathematical Software (ICMS). Charles E. Leiserson, Liyun Li, Marc Moreno Maza and Yuzhen Xie (2010) Parallel Computation of the Minimal Elements of a Poset. Proceedings of International Workshop on Parallel and Symbolic Computation (PASCO).