Multi-threaded real root isolation on multi-core architectures

(Spine title: Real Root Isolation for Polynomial System Solvers)

(Thesis format: Monograph)

by

A.B.M. Zunaid Haque

Graduate Program
in
Computer Science

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

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

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

CERTIFICATE OF EXAMINATION

Supervisor:	Examination committee:
Dr. Marc Moreno Maza	Dr. Éric Schost
	Dr. Robert E. Mercer
	Dr. Hubert Pun

The thesis by

A.B.M. Zunaid Haque

entitled:

Multi-threaded real root isolation on multi-core architectures

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

Date _____ Chair of the Thesis Examination Board

Abstract

Today, most computer algebra systems offer efficient solvers for computing the complex solutions of polynomial systems. Some of them, such as Maple, provide tools for "identifying" which of those solutions are real. These tools support many applications in areas like robotics, program verification, and dynamical system analysis, to name a few.

In this thesis, we investigate parallel algorithms for isolating the real roots of univariate equations with rational number coefficients. This type of calculation is fundamental to the computation of the real solutions of polynomial systems. Our objective is to improve performance of real root isolation on multicore processors in terms of parallelism and cache complexity, such that harder problems can be tackled on these architectures.

On multicores, the parallelization of real root isolation reduces to that of Taylor shift computations, which generalize the Pascal Triangle construction. With respect to previous works, we provide a more realistic analysis (in terms of work, span, burdened span, cache complexity) of the parallelization of this method. This leads us to develop poly-algorithms which can adapt the granularity of their parallelism dynamically depending on the local amount of work. Experimentation illustrates the effectiveness of this approach.

Keywords. Real root isolation, Taylor-shift, prefix-sum, blocking, Parallelism, Muticore architectures.

Acknowledgments

In the first place I would like to record my gratitude to my thesis supervisor Dr. Marc Moreno Maza in the Department of Computer Science at The University of Western Ontario for his supervision, advice, and guidance from the very early stage of this research as well as giving me extraordinary experiences through out the work. He consistently helped me on the way of this thesis and steered me in the right direction whenever he thought I needed it. I am indebted to him for his excellent support to me in all arenas of successful completion of this research.

Secondly, I gratefully acknowledge Dr. Changbo Chen in the Department of Computer Science at The University of Western Ontario for his advice, guidance and sharing his bright thoughts with me, which were very fruitful for shaping up my ideas and research.

Thirdly, all my sincere thanks and appreciation are extended to all the members from our Ontario Research Centre for Computer Algebra (ORCCA) lab, Computer Science Department for their constructive comments on this thesis as well as all kinds of other assistances. And I would also like to thank all the members of my thesis examination committee.

Finally, my parents deserve special mention for their inseparable support and prayers. And I am grateful to all my friends around me for their consistent encouragement.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Today, most computer algebra systems offer efficient solvers for computing the complex solutions of polynomial systems. Some of them, such as Maple, provide tools for "identifying" which of those solutions are real, provided that the input system has finitely many solutions. The case of systems with infinitely many real solutions has very limited support. This severely limits the utility of computer algebra in the areas which require exact calculation with real numbers, such as robotics, program verification, dynamical system analysis.

In "Triangular decomposition of semi-algebraic systems", C. Chen, J. H. Davenport, J. P. May, M. Moreno Maza, B. Xia and R. Xiao [4] propose an algorithm for solving systems of polynomial equations, inequations and inequalities. Under genericity assumption, this algorithm runs in singly exponential time with respect to the number of variables; improving on previously established methods. Their implementation in Maple shows promising results. However, this type of algorithm is highly demanding of computing resources, which restricts the range of the problems that an implementation can solve on traditional personal computers. To address this challenge, our research group is developing a high-performance implementation targeting clusters of multicores.

This work contributes to this project by investigating and deploying parallel algorithms for isolating the real roots of univariate equations with rational number coefficients. This type of calculation is at the core of the computation of the real solutions of polynomial systems and efficient sequential algorithms are available for this task [18, 19].

We devote our effort to the parallelization of an algorithm, called the Vincent-Collins-Akritas (VCA) Algorithm [7, 21], for isolating (that is, identifying) the real roots of a univariate polynomial with rational number coefficients (i.e.$\mathbb{Q}$[x]). As ana-

lyzed by many researchers [5, 8] the parallelization of the VCA Algorithm reduces to the parallel calculation of the *Taylor shift by 1 of a polynomial*. Although there exist asymptotically fast algorithms (running essentially in linear time with respect to the size of the input) for this task, these are not suitable for multicore architectures due to the fine-grained parallelism that these fast algorithms require. In fact, the only practical parallel algorithmic solutions for Taylor shift computation on multicore architectures are based on the construction of the Pascal Triangle.

This latter problem, and more generally the problems of parallel tableau constructions, parallel stencil computations are well known challenges in parallel processing [12]. One challenge is the *low parallelism* of these algorithms. More precisely, for an input polynomial of degree n, in the fork-join parallelism model [11], the best known complexity estimates for the parallel construction of the Pascal Triangle are obtained via a blocking strategy. If the block order is B, the work is in $\Theta(n^2)$ and the parallelism is in $\Theta(n/B)$. Moreover, for an ideal cache of Z words and cache lines of L words, the cache complexity (of the serial counterpart of this algorithm) is in $\Theta(n^2/(ZL))$, provided that B is well-chosen. Therefore, one can say that the parallelism is sub-linear, which is generally too low for multicore implementations. Another scheme for the parallel construction of the Pascal Triangle is based on a divide and conquer approach which has similar cache complexity and has the advantage of cache obliviousness. However its parallelism is even less than that of the blocking strategy.

Chapter 2 provides background material on both real root isolation and multicore programming. Chapter 3 and 4 review (respectively) the blocking strategy and divide and conquer approaches for Taylor shift computations.

Our objective is to increase the performance of real root isolation on multicore architectures in terms of parallelism and cache complexity. By increasing the performance of this operation, we expect to tackle harder problems on available parallel architectures. In fact, this idea has actually been successfully initiated in [5]. Moreover, the parallelization challenges posed by real root isolation algorithms are typical in symbolic computation. Thus, the solutions developed herein could benefit other subjects in this area.

The contributions of this thesis are three-fold:

In Chapter 5, we observe that the fork-join parallelism model does not apply to parallel Taylor shift computation for polynomials of sufficiently large degrees. This is because the fork-join parallelism model assumes that each strand works in unit time while the growth of the intermediate coefficients in the Pascal Triangle computation

invalidates this assumption. Therefore, we provide a more realistic analysis of parallel Taylor shift computation for polynomials of large degrees. This analysis shows that the parallelism is in fact higher than in the naïve analysis, but only by a constant factor. This analysis also shows that the burdened span grows asymptotically faster than the non-burdened span, a great surprise to us and which can be seen as an additional challenge for the parallelization of real root isolation on multicore architectures.

In Chapter 6, we explore the divide and conquer scheme. Based on the study conducted in Chapter 5, due to the growth of the intermediate coefficients in the Pascal Triangle construction, we observe that two recursive calls in that scheme are likely to have different work loads and different amounts of cache misses. We propose different solutions to cope with this problem and our experimentation illustrates their benefits.

In Chapter 7, we explore the blocking strategy. Based on the study conducted in Chapter 5, we turn the original blocking strategy described in [6] into a poly-algorithm which can adapt the granularity of its parallelism depending on the local size of the data. Experimentation illustrates the effectiveness of this approach.

Chapter 2

Background

We present background material used throughout. Section 2.1 is a brief introduction to techniques for finding the real roots of univariate polynomials. Descartes' famous rule of signs is discussed in Section 2.2. Section 2.3 is a review of Collin-Akritas Algorithm (VCA) for real root isolation. In Section 2.5, the Taylor Shift operation, which is the core routine for VCA, is discussed along with Horner's Method and Pascal's Triangle construction. Recall that this work focuses on the parallelization of Taylor Shift computations targeting multicore architectures. Other techniques could be considered for the parallelization of the VCA Algorithm and we will leave that for future works.

This chapter also contains a review of various aspects of multithreaded programming on multicore architectures. A brief discussion on cache memories and cache complexity is given in Section 2.6. Section 2.7 is a review of multicores as our implementation is based on this architecture. Finally Section 2.8 is a brief overview of the fork-join parallelism model and *Cilk++*. For these material on programming and architectures we follow the lecture notes of the UWO courses CS9624-4435 and CS5635-4402 which can be found at `http://www.csd.uwo.ca/~moreno/CS9624-4435-1011.html` and at `http://www.csd.uwo.ca/~moreno/CS9535-4402-1112.html`, respectively.

2.1 Real root isolation

Given a univariate polynomial f, a root finding algorithm is a numerical method for computing a value x_0 for which $f(x_0) = 0$ holds. Such an x_0 is called a *root* of the polynomial f. An important observation is that, even if the coefficients of f are numerical, one may not be able to represent x_0 exactly as a floating point number. For instance if $f(x) = x^2 - 2$ then f has two real roots $\sqrt{2}$ and $-\sqrt{2}$, none of which

being a rational number. However, for each real root x_0 of f it is always possible to find an interval $[s_0, t_0]$ with rational end points such that $[s_0, t_0]$ contains x_0 and does not contain any other real roots of f. Determining such an interval $[s_0, t_0]$ for each real root x_0 of f is referred as *isolating the real roots of* f. This leads to the following definition.

Definition 1. *Let $f \in \mathbb{R}[x]$ be a non-constant polynomial. We say that pairwise disjoint intervals $[s_0, t_0], [s_1, t_1], \ldots, [s_e, t_e]$ with rational end points* isolate the real roots *of f (or form a* real root isolation *of f) if*

1. *any real root of f is contained in one of the intervals $[s_0, t_0], \ldots, [s_e, t_e]$,*
2. *each of these intervals contains exactly one real root of f.*

Sturm's Theorem and Descartes' rule of signs [2] are the building blocks for finding the real roots location or separating them. Several algorithms are well-known, such as Vincent-Collins-Akritas Algorithm [7, 21] and Krandick's Algorithm [17]. Based on those, several more recent and more sophisticated algorithms have been introduced; they improve on certain aspects such as arithmetic complexity or memory usage. In their paper "Efficient isolation of polynomial real roots" [18], Fabrice Rouillier and Paul Zimmermann discuss memory consumption of sequential algorithms for real root isolation. These considerations, however, do not apply within the context of multithreaded programming targeting multicores. Indeed, in a parallel setting, real root isolation leads to concurrent Taylor shift computations, thus to unavoidable data duplication, which is what the Authors of [18] try to avoid. In the context of multicores, the main memory issue that we address is the minimization of cache complexity, which is not the scope of [18].

2.2 Descartes' rule of signs

Using Descartes' rule of signs we derive a bound on the number of positive real roots of a polynomial. Before stating Descartes' rule of signs as Theorem 1, we specify how to count the number of sign changes in a sequence of real numbers in Definition 2.

Definition 2. *Let $a_0, a_1, \ldots, a_n$ be a sequence of $(n+1)$ real numbers. For i, j in the interval $[0, \ldots, n]$, with $i < j$, we say that there is a* sign change from a_i to a_j *if the following two conditions hold*

(i) $a_i a_j < 0$ *and,*

(ii) for all k with $i < k < j$ we have $a_k = 0$.

Theorem 1. *Let $f \in \mathbb{R}[x]$ be a polynomial with real coefficients $a_n, a_{n-1}, \ldots, a_0$, written as:*

$$f = a_n\, x^n + a_{n-1}\, x^{n-1} + \cdots + a_0$$

where we assume $a_n a_0 \neq 0$. Let v be the number of sign changes in the sequence $a_n, a_{n-1}, \ldots, a_0$ and let r be the number of positive roots of f. Then, there exists a non-negative integer m such that we have $r = v - 2m$.

In particular, when $v = 0$ or $v = 1$ holds, we have $r = v$. This means, when the number of sign change is 0 or 1, we know the number of positive real roots is 0 or 1, respectively.

2.3 Vincent-Collins-Akritas Algorithm

Let $p \in \mathbb{Q}[x]$ be a non-constant polynomial with rational number coefficients and let $]a, b[$ be an (non-empty) open interval of $\mathbb{Q}$. In this section, we present an algorithm for isolating the real roots of p, as specified in Definition 1.

This algorithm, based on the ideas of Vincent [21] and more recently the work of Collins and Akritas [7], relies on Descartes' rule of signs. More precisely, this algorithm uses Descartes' rule to obtain an upper bound for the number of real roots of p in $]a, b[$. To justify this algorithm, we consider the following two polynomial functions:

$$f_1 : \left\{ \begin{array}{ccc}]a,b[& \to & \mathbb{R} \\ x & \longmapsto & p(x) \end{array} \right. \quad \text{and} \quad f_2 : \left\{ \begin{array}{ccc}]0,\infty[& \to & \mathbb{R} \\ x & \longmapsto & \overline{p}(x) \end{array} \right. \tag{2.1}$$

where $\overline{p}(x) = (x+1)^d p(u(x))$ and $u(x) = \frac{ax+b}{x+1}$. Observe that $u'(x) = \frac{a-b}{(x+1)^2}$. Since we have $a - b < 0$, the function u is strictly decreasing on $]0, \infty[$. Moreover we have:

$$\lim_{x \to 0} u(x) = b \quad \text{and} \quad \lim_{x \to +\infty} u(x) = a.$$

Therefore, the function u realizes a 1-to-1 map from $]0, \infty[$ onto $]a, b[$. Next we observe that $\overline{p}(x)$ is a polynomial, thanks to the multiplication of $p(u(x))$ by $(x+1)^d$, which clears out denominators in $p(u(x))$. Now we note the following relation, for every $x \in]0, \infty[$,

$$\overline{p}(x) = 0 \quad \Longleftrightarrow \quad p(u(x)) = 0. \tag{2.2}$$

Thus x is a positive root of the polynomial $\overline{p}$ if and only if $u(x)$ is a root of the polynomial p in $]a, b[$. Therefore, by applying Descartes' rule of signs to the

polynomial $\overline{p}$, we count the number of real roots of p in $]a, b[$. This proves the correctness of Algorithm 1.

Algorithm 1: `BoundNumberRootsVCA`$(p,]a, b[)$

Input: p squarefree polynomial $\in \mathbb{Q}[x]$ and $a \leq b$ are in $\mathbb{Q}$.
Output: Number of roots of p in the interval $]a, b[$.

1: $\overline{p} = (x+1)^d p(\frac{ax+b}{x+1})$ where d is the degree of p;
2: From $\overline{p}$ determine the coefficient sequence;
3: **return** the number of sign variations in the coefficient sequence.

In Algorithm 1, the call `BoundNumberRootsVCA`$(p,]a, b[)$, uses Descartes' rule to obtain an upper bound for the number of real roots of p in $]a, b[$. In the call `RootIsolateAuxVCA`$(p,]a, b[)$, Algorithm 2 uses Algorithm 1 to determine a real root isolation (in the sense of Definition 1) of a square free polynomial $p \in \mathbb{Q}[x]$. For the interval $]a, b[$, three cases need to be considered for the number of sign variations v:

(i) $v = 0$, thus no roots in $]a, b[$,
(ii) $v = 1$, thus a single root in $]a, b[$ and
(iii) $v > 1$, in which case we cannot conclude on the number of real roots of p in $]a, b[$.

For Case (iii), the interval $]a, b[$ is divided into two equal parts and for each of them Algorithm 2 is called recursively. Also, Algorithm 2 checks whether the mid-point of the interval $m = \frac{a+b}{2}$ is itself a root or not.

Algorithm 2: `RootIsolateAuxVCA`$(p,]a, b[)$

Input: p squarefree polynomial $\in \mathbb{Q}[x]$ and $a \leq b$ in $\mathbb{Q}$.
Output: An interval decomposition of real roots of p in the range]a,b[.

1: v = Number of sign variations;
2: Calculate v from `BoundNumberRootsVCA`$(p,]a, b[)$;
3: **if** $v = 0$ **then**
4: | return ϕ
5: **else if** $v = 1$ **then**
6: | return]a,b[
7: **else**
8: | $m = \frac{a+b}{2}$ $res \longleftarrow \phi$;
9: | **if** $p(m) = 0$ **then**
10: | | $res \longleftarrow \{\{m\}\}$;
11: | **return** `RootIsolateAuxVCA`$(p,]a, m[) \cup res \cup$ `RootIsolateAuxVCA`$(p,]m, b[)$.

Algorithm 3, `RootIsolateVCA`(p) is the top-level procedure, which calls Algorithm 2 after determining an initial interval containing all the real roots of the input polynomial. This initial interval can be obtained from a root bound like the *Cauchy bound*.

Algorithm 3: `RootIsolateVCA`(p)

Input: p squarefree polynomial $\in \mathbb{Q}[x]$.
Output: An interval decomposition of real roots of p.
1: Calculate a strict bound H on the absolute value for every roots of p;
2: **return** `RootIsolateAuxVCA`$(p,]-H, H[)$

There are different ways to calculate the strict bound H for a polynomial $p = \sum_{i=0}^{d} c_i x^i$. **Cauchy bound** gives $H = \frac{1}{c_d} \sum_{i=0}^{d} |c_i|$. For more details on root estimates (bounds, etc.) and for a termination proof of Algorithm 3, see the Phd Thesis of Jeremy R. Johnson [15].

We summarize some important complexity results on the VCA Algorithm and related routines. As before, let $p \in \mathbb{Q}[x]$ be a squarefree polynomial of degree d. Let L be an upper bound of the bit size of the coefficients of p. In [20], Arnold Schönhage, with his *splitting circle method*, proves that isolating the real roots of p could be done within $O^{\sim}(Ld^3)$ bit operations. A more practical algorithm, with a slightly higher complexity of $O^{\sim}(L^2d^3)$, is proposed by Michael Sagraloff in [19]. Finally, in [9], Arno Eigenwillig, Vikram Sharma and Che K. Yap proved that the depth of the recursion tree of Algorithm 5 is in $O(d(L + \log(d)))$.

2.4 A variant of the VCA Algorithm

It is not hard to see that the dominant cost in the VCA Algorithm is the computation of the polynomial

$$\overline{p} = (x+1)^d p\left(\frac{ax+b}{x+1}\right) \tag{2.3}$$

in Algorithm 1. It is, therefore, natural to dedicate one's effort toward this computation. The polynomial $\overline{p}$ depends on three parameters, namely: p, a, b. A first step toward optimizing this computation is to reduce the number of parameters while maintaining (or reducing) the same algebraic complexity. There is a standard way to do this, see for instance [18] among other references. With Algorithms 4 and 5, we follow here the presentation of Changbo Chen, Marc Moreno Maza and Yuzhen Xie in their paper [5].

Clearly, the problems of isolating the real roots of p and that of counting of real roots of p are essentially the same problem. Thus, an algorithm solving one of these problems can be adapted to solve the other. Since the output of the counting problem is simpler, Algorithms 4 and 5 solve this counting roots problem.

Algorithm 4: `realRoots`(p, k)

Input: An univariate squarefree polynomial $p \in \mathbb{Q}[x]$ of degree d, an integer $k \geq 0$ such that the absolute value of any real root of p is less than or equal to 2^k.

Output: The number of real roots of p.

1: **if** $x \mid p$ **then**
2: $\quad m = 1$
3: **else**
4: $\quad m = 0$
5: $p_1 = p(2^k x)$;
6: $p_2 = p_1(-x)$;
7: $m' =$ `RootsInZeroOne`(p_1);
8: $m = m +$ `RootsInZeroOne`(p_2);
9: **return** $m + m'$.

In Algorithm 4 we transform the polynomial $p(x)$ into the polynomial $p_1(x)$ given by $p_1(x) = p(2^k x)$, so that the roots of $p(x)$ in the interval $]-2^k, 2^k[$ are in 1-to-1 correspondence with the roots of $p_1(x)$ in the interval $]-1, 1[$. We prove this fact in the following way: Consider two polynomial functions

$$f_1 : \left\{ \begin{array}{ccc}]-2^k, 2^k[& \rightarrow & \mathbb{R} \\ x & \longmapsto & p(x) \end{array} \right. \quad \text{and} \quad f_2 : \left\{ \begin{array}{ccc}]0, 1[& \rightarrow & \mathbb{R} \\ x & \longmapsto & p_1(x) \end{array} \right. \tag{2.4}$$

where $p_1(x) = p(u(x))$ and $u(x) = 2^k x$. Observe that $u'(x) = 2^k$, thus u is strictly increasing on $]0, 1[$. Moreover we have:

$$\lim_{x \to 0, x > 0} u(x) = 0 \quad \text{and} \quad \lim_{x \to +1, x < +1} u(x) = 2^k.$$

Therefore, the function u realizes a 1-to-1 map from $]0, 1[$ onto $]0, 2^k[$. Now, we note the following relation, for every $x \in]0, 1[$,

$$p_1(x) = 0 \quad \Longleftrightarrow \quad p(u(x)) = 0. \tag{2.5}$$

Thus, some $x \in]0, 1[$ is a root of the polynomial p_1 if and only if $u(x)$ is a root of

the polynomial p in $]0, 2^k[$. Similarly, one can check that $x \in]-1, 0[$ is a root of the polynomial p_2 if and only if $u(-x)$ is a root of the polynomial p in $]-2^k, 0[$. Finally, $x = 0$ is a root of p if and only if x divides p. Combining these three observations proves the correctness of Algorithm 4.

The main subroutine of Algorithm 4 is Algorithm 5. The most expensive operation is the *Taylor shift computation*, which substitutes x with $x + 1$ in p_1 at Lines 2 and 7. Section 2.5 is dedicated to the Taylor shift computation.

Algorithm 5: `RootsInZeroOne`(p)

Input: An univariate squarefree polynomial $p \in \mathbb{Q}[x]$ of degree d.

Output: The number of real roots of p in $]0, 1[$.

1: $p_1 = x^d p(1/x)$;
2: $p_2 = p_1(x + 1)$; `/* Taylor shift */`
3: Let v be the number of sign variations of the coefficients of p_2;
4: **if** $v \leq 1$ **then**
5: | return v
6: $p_1 = 2^d p(x/2)$;
7: $p_2 = p_1(x + 1)$; `/* Taylor shift */`
8: **if** $x \mid p_2$ **then**
9: | $m = 1$
10: **else**
11: | $m = 0$
12: $m' =$ `RootsInZeroOne`(p_1);
13: $m = m +$ `RootsInZeroOne`(p_2);
14: **return** $m + m'$.

In Algorithm 5, at Line 1 we define $p_1(x) := x^d p(1/x)$ so that the roots of $p(x)$ in the range $]0, 1[$ are in a 1-to-1 correspondence with the roots of $p_1(x)$ in the range $]1, +\infty[$. The justification of this is similar to that of Algorithm 4. At Line 2, we compute $p_2(x)$ (the Taylor shift by 1 of $p_1(x)$), so that now the roots of $p(x)$ in the range $]0, 1[$ are in a 1-to-1 correspondence with the roots of $p_2(x)$ in the range $]0, +\infty[$. Thus we can apply Descartes' rule to $p_2(x)$ in order to estimate the number of real roots of p in the range $]0, 1[$. More precisely, let v be the number of sign changes in $p_2(x)$. We know from Theorem 1 that if $v \leq 1$ holds then p admits exactly v real roots in the range $]0, 1[$. This fact is implemented at Line 4. If $v > 1$ holds, then we search for real roots of p in $]0, 1/2[$, in $]1/2, 1[$ at $x = 1/2$. This is done via the

polynomials $p_1(x)$ and $p_2(x)$ (defined at Lines 6 and 7), by two recursive calls (at Lines 12 and 13) and by testing whether x divides p_2 (at Line 8).

2.5 Taylor shift

The *Taylor shift* is a core routine among real root isolation algorithms. As we saw before, the VCA Algorithm reduces the problem of isolating the real roots of an univariate squarefree $p(x) \in \mathbb{Q}(x)$ to that of computing the coefficients of $p(x+1)$ in the monomial basis. This latter computation is referred to as the *Taylor shift* of p by 1.

More generally, let $p = \sum_{0 \le i \le n} c_i x^i \in \mathbb{Q}(x)$ and let $a \in \mathbb{Q}$. Computing the coefficients of $p(x+a)$ in the monomial basis, say $g_0, \ldots, g_n \in Z$, such that we have

$$p(x+a) = \sum_{0 \le k \le n} g_k \, x^k,$$

is refered as the *Taylor shift* of p by a. The following proposition states an expression of the coefficients $g_0, \ldots, g_n \in \mathbb{Q}$ as functions of the coefficients $c_0, c_1, \ldots, c_n$ and a.

Proposition 1. *With the above notation, we have for each $i = 0 \cdots n$,*

$$g_i = \sum\nolimits_{j=0}^{n-i} c_{i+j} \binom{i+j}{i} a^j. \tag{2.6}$$

Proof. Recall $p(x) = \sum_{i=0}^{n} c_i x^i$ and $g(x) = \sum_{i=0}^{n} g_i x^i$ where $g(x) = p(x+a)$. We have:

$$p(x+a) \;=\; c_0 + c_1(x+a) + c_2(x+a)^2 + \cdots + c_n(x+a)^n. \tag{2.7}$$

From the above expression, we can deduce g_0, g_1, g_n:

$$\begin{array}{rcl} g_0 & = & c_0 + c_1 a + c_2 a^2 + \cdots + c_n a^n \\ g_1 & = & c_1 + 2ac_2 + 3c_3 a^2 + \cdots + n c_n a^{n-1} \\ & \vdots & \\ g_n & = & c_n \end{array} \tag{2.8}$$

For the general case, that is for g_i, we need to understand what is the contribution of each term from the following expression:

$$c_i(x+a)^i + c_{i+1}(x+a)^{i+1} + \cdots + c_{i+j}(x+a)^{i+j} + \cdots + c_n(x+a)^n. \tag{2.9}$$

Recall the binomial formula:

$$(u+v)^s \;=\; \sum_{k=0}^{s} \binom{s}{k} u^k v^{s-k}. \tag{2.10}$$

This shows that the coefficient of x^i in $c_{i+j}(x+a)^{i+j}$ is $c_{i+j}\binom{i+j}{i}a^j$. Therefore, we have:

$$g_i \;=\; \sum_{j=0}^{n-i} c_{i+j} \binom{i+j}{i} a^j. \tag{2.11}$$

□

An important special case is $a = \pm 1$ [22]. There are classical methods for computing the coefficients $g_0, \ldots, g_n$: the famous *Horner Method* (see Section 2.5.1), *Shaw and Traub's method* (see [22]) and *multiplication-free methods* such as the *Pascal Triangle Method* (see Section 2.5.2). This last is at the base of two parallel algorithms presented in [6] and which we review in Sections 3.1 and 4.1.

2.5.1 Horner's method for Taylor shift by a

Let p and g be as above. Horner's rule relies on the following re-combination of terms:

$$\begin{aligned} g(x) &= p(x+a) \\ &= c_0 + (x+a)\,(c_1 + \cdots + c_n\,(x+a)^{n-1}) \end{aligned} \tag{2.12}$$

whose purpose is to evaluate any univariate polynomial of degree n within $2\,n$ additions and multiplications. Let us write:

$$g^n(x) = g(x) \;\text{ and }\; g^{n-1} = c_1 + \cdots + c_n\,(x+a).$$

Then we have:

$$g^n(x) = c_0 + (x+a)g^{n-1}(x). \tag{2.13}$$

Therefore, if $g^{n-1}(x)$ has been expanded on the monomial basis, we can deduce a monomial basis expansion of g^n. The number $T(n)$ of arithmetic operations is thus given by

$$T(n) = T(n-1) + 2n + n,$$

which implies that Horner's rule requires $3/2n^2 + 3/2n + 1$ additions and multiplications. See [22] for details.

2.5.2 Taylor shift by 1 via Pascal's Triangle construction

From now on we restrict ourselves to the case $a = 1$ which is what we need for real root isolation. Formula (2.13) becomes:

$$g^n(x) = c_0 + (x+1)g^{n-1}(x).$$

This shows that the term of order i in $g^n(x)$ is obtained by adding the term of degree i from $g^{n-1}(x)$ (if any) and the term of degree $i-1$ from $g^{n-1}(x)$. This gives rise to a multiplication-free algorithm which boils down constructing a Pascal Triangle (see Figure 2.1). The number of additions necessary to compute $g_0, \ldots, g_n$ is then equal to the number of entries that are filled in the triangle, which is $(n+2)(n+1)/2$.

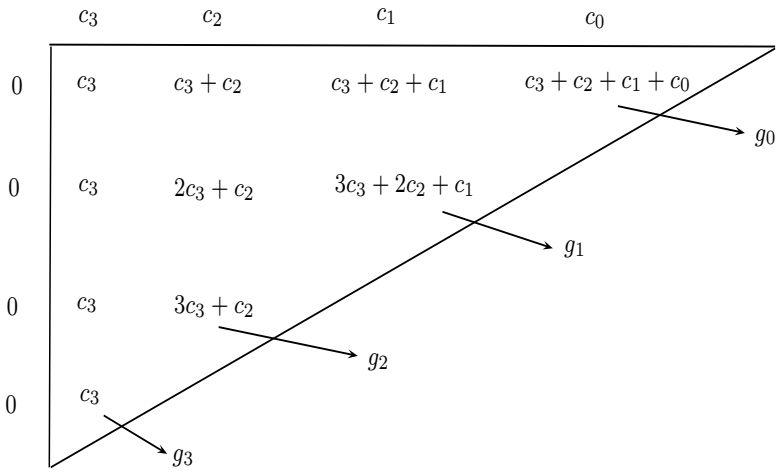


Figure 2.1: Pascal Triangle.

The Pascal Triangle can be computed in different ways. Following [6] we consider two schemes that we call *divide and conquer* and *blocking*. The former is discussed in Chapters 3 and 6 while the latter in Chapters 4 and 7.

2.6 Cache memories and cache complexity

2.6.1 Cache memories

A cache is a region of relatively small memory used by the central processing unit (CPU) of a computer. The access time to this cache memory is very fast compared to the access time of the main memory. Data is brought into the cache memory by the CPU before it can be used by the CPU registers. Once in cache, data can be

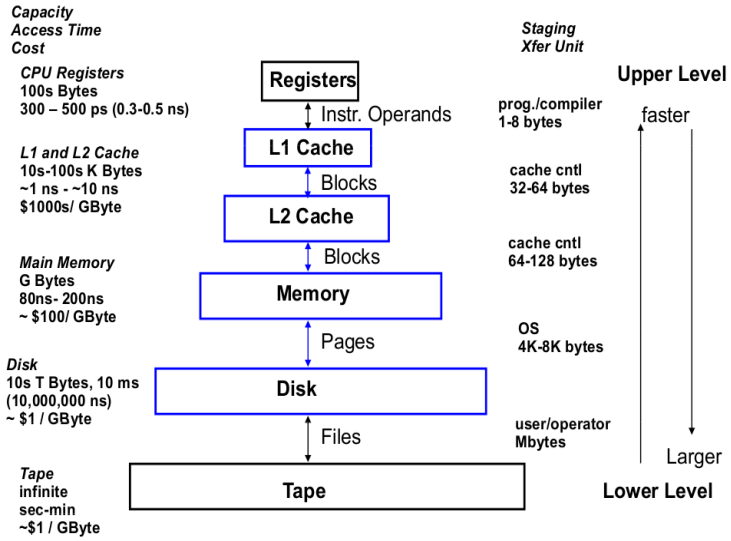


Figure 2.2: Memory hierarchy in a computer

reused for several computations by the CPU. When a data item required by the CPU is found in the cache memory, we say that we have a *cache hit*, otherwise we have *cache miss*.

Using cache memory effectively can reduce memory access time spectacularly. C or FORTRAN programs implementing fundamental operations such as matrix multiplication can be accelerated by two orders of magnitude[1] *simply* by optimizing the use of cache memory. Modern desktops and laptops have typically two levels of cache memory.

L1-cache: The L1-cache is the fastest and it usually comes embedded within the processor chip itself. The L1 cache typically ranges from 8KB to 64KB and uses high-speed SRAM (static RAM) instead of the slower and cheaper DRAM (dynamic RAM) used for main memory.

L2-cache: The L2-cache is bigger than L1-cache and it sits between L1 and RAM. It is typically 64KB to 4MB.

Multi-level caches generally operate by checking for data in the L1 cache; if found, the processor proceeds at high speed. If the data is not in L1 cache, the next larger cache (L2) is checked before main memory is checked.

Data is moved between memory levels in *cache lines*, also called *blocks* and which typically hold 64 to 256 bytes. This implies that when the CPU loads a value (say

[1] `http://www.csd.uwo.ca/ moreno//Publications/SHARCNET_Tutorial.pdf`

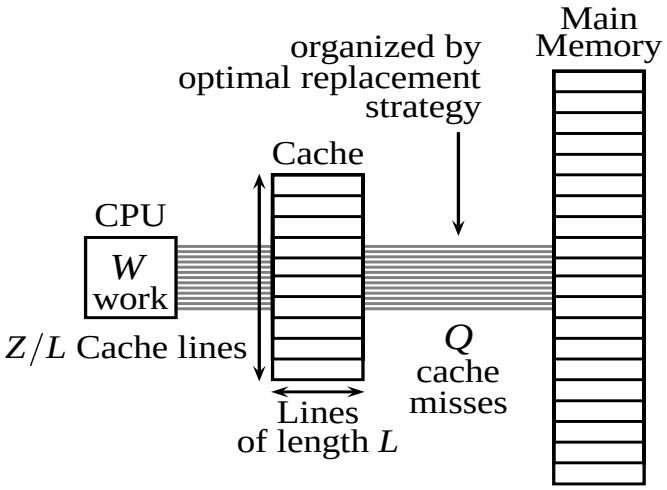


Figure 2.3: Ideal cache model

an integer of a floating point number) from a lower level memory to a higher level memory, several variables are actually loaded. This mechanism suggests that there are two properties for a program to perform well toward avoiding unnecessary and costly memory operations (loads and stores).

Temporal locality: refers to the reuse of specific data within relatively small time durations.

Spatial locality: refers to the use of data elements within relatively close storage locations.

2.6.2 Cache complexity

The ideal cache model, introduced in [10], assumes a computer system with a two-level memory hierarchy where the first level is an ideal (data) cache of Z words and the second level is an arbitrarily large main memory. The cache is partitioned into Z/L *cache lines* where L is the length of each cache line, in other words, each cache line can hold L consecutive words which always move together between the cache and main memory. Cache designers usually use $L > 1$ to achieve *spatial locality*. It is generally assumed that the cache Z is much larger than L. More precisely,

$$Z = \Omega(L^2).$$

This type of cache is called *tall* cache, which is always the case in practice.

When a word required by the processor is found in cache, we say (as in the previous section) that we have a *cache hit*. Otherwise, we say that we have a *cache miss* and the line can be fetched in any available block into in the cache. This mapping from main memory into the cache is called *full associativity*. If the cache is full a cache line must be evicted. In the ideal cache model, the cache line whose next access is the furthest in the future is replaced [1]: this is called an *optimal off-line strategy of replacing*.

For an algorithm with an input of size n, the ideal-cache model uses two complexity measures:

1. the *work complexity* $W(n)$, which is its conventional running time in a RAM model.

2. the *cache complexity* $Q(n; Z, L)$, the number of cache misses it incurs (as a function of the size Z and line length L of the ideal cache).

When Z and L are clear from context, we simply write $Q(n)$ instead of $Q(n; Z, L)$.

An algorithm is said to be *cache aware* if its behavior (and thus performance) can be tuned (and thus depend on) on the particular cache size and line length of the targeted machine. Otherwise the algorithm is *cache oblivious*.

2.7 Multicore architecture

A *multi-core processor* is an integrated circuit to which two or more individual processors (called cores) have been attached. In a many-core processor the number of cores is large enough that traditional multi-processor techniques are no longer efficient. Cores on a multi-core implement the same architecture features as single-core systems such as instruction pipeline parallelism (ILP), vector-processing, SIMD, and multi-threading. Many applications do not yet realize large speedup factor: parallelizing algorithms and software is a major on-going research area.

There are two important issues related to multicore architectures that may significantly reduce performances.

True sharing cache misses occur whenever two processors access the same data word. True sharing requires the processors involved to explicitly synchronize to ensure program correctness. Programs with high temporal locality tend to have less true sharing.

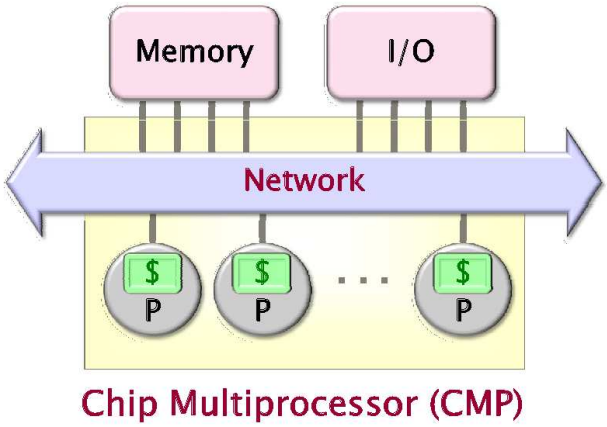


Figure 2.4: Multicore architecture

False sharing occurs whenever different processors use different data that happen to be co-located on the same cache line. Enhancing spatial locality often minimizes false sharing.

High levels of false or true sharing and synchronization can easily overwhelm the advantage of parallelism. (See *Data and Computation Transformations for Multiprocessors* by J.M. Anderson, S.P. Amarasinghe and M.S. Lam `http://suif.stanford.edu/papers/anderson95/paper.html`.)

Another significant cause of poor performance in multi-threaded programs on multicore architectures is the fact that access to the main memory (RAM) is serialized. More precisely, the RAM memory controller can serve only one load or store request at a time. However, CPUs in a multicore can issue those requests faster than the memory controller can process them. As a result of this *memory contention* phenomenon, the performance of a multi-threaded program can be dramatically reduced up to a point that the potential benefits of concurrent execution are annihilated.

2.8 The fork-join parallelism model and Cilk++

2.8.1 The Cilk++ concurrency platform

`Cilk` has been in development since 1994 at the Massachusetts Technology Institute (MIT) by *Prof. Charles E. Leiserson* and his research group, in particular by *Matteo Frigo*. Besides being used for research and teaching, `Cilk` was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess. Over the years,

Cilk has run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon. From 2007 to 2009 Cilk has lead to Cilk++, developed by Cilk Arts, an MIT spin-off, which was acquired by Intel in July 2009 and became Cilk Plus, see http://www.cilk.com/ Today, Cilk++ can be freely downloaded at

http://software.intel.com/en-us/articles/download-intel-cilk-sdk/

Cilk is still developed at MIT by Prof. Charles E. Leiserson and his research group, see the page:

http://supertech.csail.mit.edu/cilk/

Cilk++ (resp. Cilk) is a small set of linguistic extensions to C++ (resp. C) supporting *fork-join parallelism*. In the Cilk++ program below, the named *child* function cilk_spawn fib(n-1) may execute in parallel while its *parent* executes fib(n-2). The Cilk++ keywords cilk_spawn and cilk_sync grant permissions for parallel execution. They do not command parallel execution.

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  x = cilk_spawn fib(n-1);
  y = fib(n-2);
  cilk_sync;
  return x+y;
}
```

2.8.2 The fork-join parallelism model

A simple theoretical model for the parallel execution of a Cilk++ program is a DAG (directed acyclic graph) where

- each vertex represents a *strand*, that is, a sequence of consecutive instructions, to be executed serially and
- each edge indicates a chronological dependency between two strands.

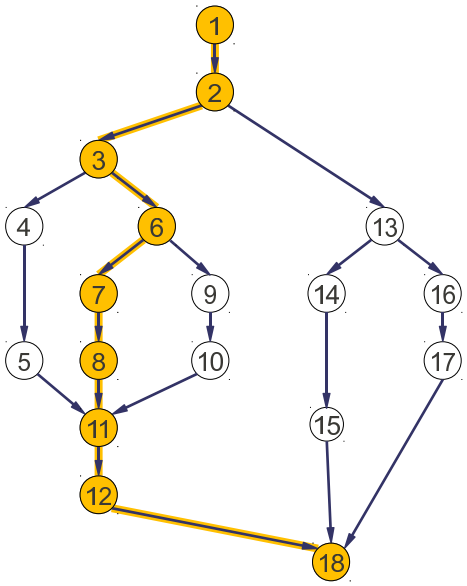


Figure 2.5: A directed acyclic graph (DAG) representing the execution of a multithreaded program. Each vertex represents a strand and each edge represents a dependency between two strands.

This DAG is called an *instructions stream DAG*. See Figure 2.5 for an example and [3] for details.

Three performance measures are naturally associated to a `Cilk++` and, more generally, to an instruction stream DAG.

T_p is the minimum running time on p processors

T_1 is called the *work*, that is, the sum of the number of instructions at each vertex in the DAG,

T_∞ is the minimum running time with infinitely many processors, called the *span*.

Assuming that all strands run in unit time (assuming also no cache issues and no inter-processor costs) the longest path in the DAG from the initial strand to a final strand is T_∞. For this reason, T_∞ is also referred to as the *critical path length*.

Since, in the best case, p processors can do p work per unit of time, we have: $T_p \geq T_1/p$, which is often referred as the *work law*.

Since $T_p < T_\infty$ would contradict the definitions of T_p and T_∞, we have $T_p \geq T_\infty$, which is often referred as the *span law*.

The quantity T_1/T_p is called the *speedup on p processors*. A parallel program execution can have:

- *linear speedup*: $T_1/T_P = \Theta(p)$,
- *superlinear speedup*: $T_1/T_P = \omega(p)$ (not possible in this model, though it is possible in others), and
- *sublinear speedup*: $T_1/T_P = o(p)$.

By definition, the *parallelism* of an instruction stream DAG is the ratio work to span, namely T_1/T_∞. This represents the average number of strands which can be run concurrently along the critical path.

2.8.3 Work-stealing scheduling

The Cilk++ runtime system has a work stealing scheduler which is reliable in the sense that it can dynamically and automatically exploit an arbitrary number of available processor cores nearly optimally. When the Cilk++ runtime system starts up, it allocates as many system threads, called *workers* for the processors. When any subroutine is spawned, the activation frame is pushed to the bottom of the stack of the corresponding processor. When the child is returned to its parent, the parent's activation frame is popped off the stack. During the execution of the program, if any worker runs out of work, it "steals" work from (the top of) another worker's stack.

Assume that, for an application running with P processors:

- each strand executes in unit time,
- for almost all "parallel step" there are at least P strands to run, and
- each processor is either working or stealing.

Then the `Cilk++` work-stealing scheduler achieves an expected running time as

$$T_P \leq T_1/P + O(T_\infty)$$

.

If the parallelism T_1/T_∞ is so large that it sufficiently exceeds P, then this bound provides a nearly perfect linear speed up. Assuming $T_1/T_\infty \gg P$, or equivalently $T_1/P \gg T_\infty$, the inequality leads to $T_P \approx T_1/P$. `Cilk++` estimates T_p as $T_p = T_1/p + 1.7S_b$ where S_b is the *burdened span*, that is, 15000 instructions times the number of spawns along the critical path. See Sections 2.8.4 and 2.8.5 for details.

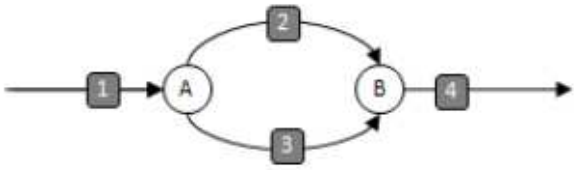


Figure 2.6: Cilk++ execution model dag.

2.8.4 Cilk++ execution model

This section is essentially adapted from the `Cilk++` Programming guide which can be found at `http://software.intel.com/en-us/articles/intel-cilk-plus/`. Recall that the instruction stream DAG of a `Cilk++` program does not depend on the number of processors. The execution model describes how the runtime scheduler maps strands to workers. Consider the following `Cilk++` program fragment:

```
do_init_stuff();        // execute strand 1
cilk_spawn func3();     // spawn strand 3 (the "child")
do_more_stuff();        // execute strand 2 (the "continuation")
cilk_sync;
do_final_stuff;         // execute strand 4
```

A DAG for the code is showed in Figure 2.6.

If there are more than one worker available, there are two ways this program may execute:

1. the entire program may execute on a single worker, or

2. the scheduler may choose to execute strands (2) and (3) on different workers.

If there is a worker available, then strand (2) (the "**continuation**", we refer to this as the continuation since this strand continues from where the parent strand was spawned) may execute on the current (parent) worker or on a different worker (the latter situation is referred to as "work stealing").

In Figure 2.7 we have shown that the work has been stolen from the parent worker and a second worker will begin executing the continuation, strand (2). The first worker (parent) will proceed to the synchronization at (B). Here, we indicate the

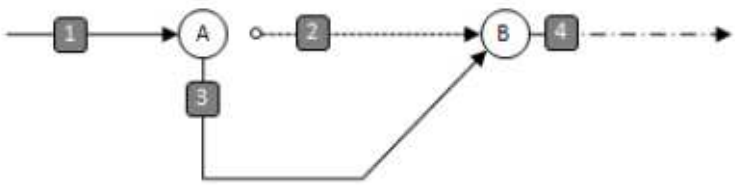


Figure 2.7: Cilk++ execution model dag.

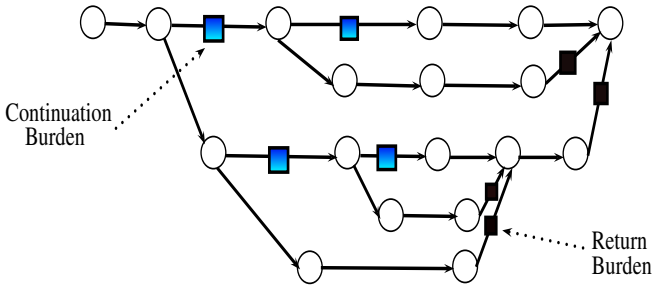


Figure 2.8: Burdened DAG due to continuation and return burden

second worker by illustrating strand (2) with a dotted line. After synchronization, strand (4) may continue on either worker.

2.8.5 Burdened parallelism

When work is stolen, there is a cost for migrating the stolen task's work set. There is also a cost for the continuation and return edges (in the DAG) which reflects overheads due to scheduling. The performance analyzer `Cilkview` [13] assumes that, for each spawn, a burden of 15,000 instruction cycles is created due to locks, malloc, cache warmup, etc.

Figure 2.5 shows burden on each continuation edge in the DAG. The ***burdened span*** is the longest path in the burdened dag. The migration cost can incorporate the bound $T_P \leq T_1/P + O(T_\infty)$ as

$$T_P \leq T_1/P + 2\delta\widehat{T_\infty}$$

where δ is the span coefficient and $\widehat{T_\infty}$ is the burdened span while the program is running on P processor. The burden parallelism is computed by $T_1/\widehat{T_\infty}$. See [13] for details.

2.8.6 Cilk_for

This section is also essentially repeating the `Cilk++` Programming guide which can be found at `http://software.intel.com/en-us/articles/intel-cilk-plus/`. A *cilk_for* loop is a replacement for the normal `C++ for loop` that permits loop iterations to run in parallel. A Sample *cilk_for* loop is like:

```
cilk_for (int i = begin; i < end; i += 2){
     f(i);
}
```

Using *cilk_for* is not the same as spawning each loop iteration. In fact, the `Cilk++` compiler converts the loop body into a function that is called recursively using a divide and conquer strategy, which allows the `Cilk++` scheduler to provide significantly better performance. The difference between a *cilk_for* and a `for` loop spawning each loop iteration can be seen clearly using instruction stream DAGs.

First, the DAG for a *cilk_for* provided in Figure 2.9 (assuming $N = 8$ iterations). The numbers labeling the strands indicate which loop iteration is handled by each strand. Note that at each division of work, half of the remaining work is done by the the child thread and half by the continuation. Importantly, the overhead of both the loop itself and of spawning new work is divided evenly along with the cost of the loop body.

If each iteration takes the same amount of time T to execute, then the span is $log_2(N)\,T$, or $3T$ for 8 iterations. The runtime behavior is well-balanced regardless of the number of iterations or workers.

But, in the case of spawning loop iterations, the work is not well balanced, because each child does the work of only one iteration before incurring the scheduling overhead inherent to entering a sync. For a short loop, or a loop in which the work in the body is much greater than the control and spawn overhead, there will be little measurable performance difference. However, for a loop of many cheap iterations, the overhead cost will overwhelm any advantage provided by parallelism. The DAG for a serial loop that spawns each iteration is shown in Figure 2.10.

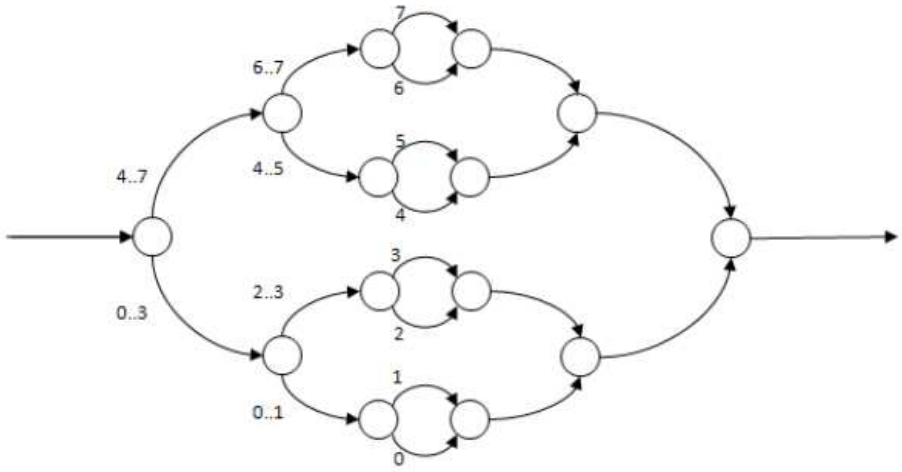


Figure 2.9: DAG for *cilk_for*.

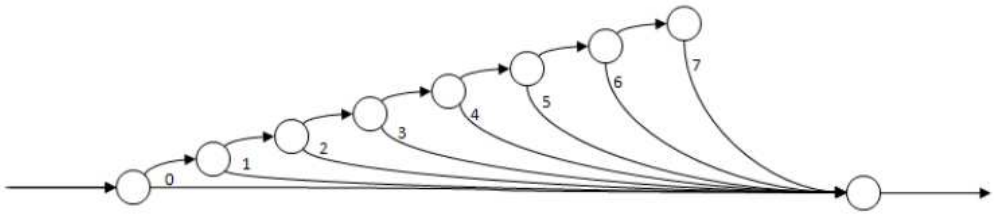


Figure 2.10: DAG for *cilkspawn*.

Chapter 3

Divide and Conquer Taylor shift: Static approach

Taylor shift computation, based on the Pascal Triangle construction, can be performed using different schemes. We consider two schemes that we call *divide and conquer* and *blocking*, following [5]. This chapter is dedicated to the former and the next chapter to the latter. Both chapters recall these schemes and provide complexity estimates for work, span and cache misses. These results already appear in [5] but with sketches of proof, whereas we make these proofs more complete.

We say that the algorithms of this chapter and the next one follow a "static approach" in the sense that they are not able to adapt themselves to the phenomenon of intermediate expression swell, studied in Chapter 5. On the contrary, the algorithms of Chapter 6 and 7 are *adaptive* (or *dynamic*), as they are able to dynamically change the granularity of their parallelism in order to improve performance.

3.1 Implementation scheme for static divide and conquer

Recall from Section 2.5.2 that the Taylor shift by 1 of a polynomial can be obtained via a Pascal Triangle construction. One can observe from Figure 3.1 that a Pascal Triangle can be constructed in a recursive manner: first, computing the elements in the top-left square region, then computing the elements of the two triangular regions. Moreover, observe that the two triangular regions can be evaluated concurrently. This recursive way of constructing a Pascal Triangle is called the *Divide and Conquer* method. This method uses a base case to determine until which point the triangular

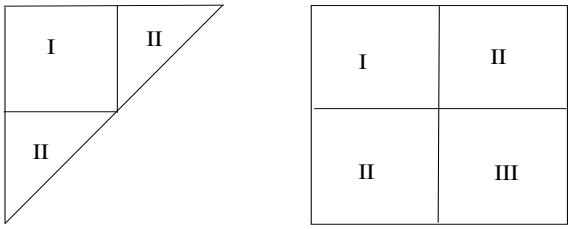


Figure 3.1: Divide and conquer Taylor shift.

or the square regions will be divided into smaller regions. If this base size remains the same for the entire computation, then this divide and conquer process can be referred to as *Static divide and conquer*. We will focus only on the static approach. Later in Chapter 6 we discuss the dynamic approach.

Figure 3.1 shows Static divide and conquer in a simplified fashion. The Pascal Triangle is divided into 3 regions, one tableau (region I) , and two triangles (region II) . These two triangle regions can work parallel. Tableau region is further divided into four regions I, II, II, III respectively. For tableau, the regions denoted as II can be computed concurrently. When they reach to base case, they stop doing division and everything is computed in a sequential manner. For static divide and conquer, base case should have to be at least of size 2, means it must contain at least 2 elements to compute.

3.1.1 Algorithms for static divide and conquer

Let B be the order of a block, $B > 0$. For a polynomial of degree d, we have n inputs for the Pascal Triangle, where $n = d + 1$.

Algorithm 6, `taylorShiftGeneral` does the task of recursively dividing each triangle into one square and two triangular regions. If $n \leq B$ then the whole computation is done serially shown at Lines 1 and 2. In Line 5, Algorithm 7 is called which computes the square part. Lines 6 and 7 are two recursive calls with upper and lower triangles. When the triangles reach the base case this means no further division is required. Algorithm 9 is then called.

Algorithm 7, `tableauConstruction`, is for recursively dividing the square region (Tableau) until base case is reached. Like Algorithm 6, Algorithm 7 is also a recursive algorithm where a base case is computed by Algorithm 8.

Algorithm 8, `tableauBaseInplace` works when the square region reaches the base

Algorithm 6: `taylorShiftGeneral`$(p[0 \dots n-1], q[0 \dots n-1], B)$

Input: $p[0 \dots n-1]$ is the coefficient array (dense representation) of an univariate polynomial $p(x)$ of degree d where $n = d+1$ ($a[i]$ is the coefficient of the term of degree i); $q[0 \dots n-1]$ is another array of length n, where all coefficients are initially zero; both $p[0 \dots n-1]$ and $q[0 \dots n-1]$ are overwritten; B is the base size (threshold).

Output: Coefficient array of the polynomial $p(x+1)$ stored in the array $p[0 \dots n-1]$ such that $a[i]$ is the coefficient of the term of degree i.

1: **if** $n \leq B$ **then**
2: `taylorShiftBase`$(p[0, \dots, n-1], q[0, \dots, n-1])$;
3: **else**
4: $m = \frac{n+1}{2}$;
5: `tableauConstruction`$(p[0 \dots m-1], q[0 \dots m-1], B)$;
6: **spawn** `taylorShiftGeneral`$(p[0 \dots m-1], q[m \dots n-1], B)$;
7: **spawn** `taylorShiftGeneral`$(p[m \dots n-1], q[0 \dots m-1], B)$;
8: **sync**;

Algorithm 7: `tableauConstruction`$(p[0 \dots m-1], q[0 \dots n-1], B)$

Input: Array $p[0 \dots n-1]$ of size m and array $q[0 \dots n-1]$ of size n; initially $p[0 \dots m-1]$ and $q[0 \dots n-1]$ contain the input coefficients; $p[0 \dots m-1]$ and $q[0 \dots n-1]$ are overwritten during the computations; B is the base size (threshold).

Output: Result of right edge of Tableau is stored on array $p[0 \dots m-1]$ and down edge of tableau is stored on array $q[0 \dots n-1]$.

1: **if** *($m \leq B$)* AND *($n \leq B$)* **then**
2: `tableauBaseInplace`$(p[0 \dots m-1], q[0 \dots n-1])$;
3: **else**
4: $i = \frac{m+1}{2}$;
5: $j = \frac{n+1}{2}$;
6: `tableauConstruction`$(p[0 \dots i-1], q[0 \dots j-1], B)$;
7: **spawn** `tableauConstruction`$(p[0 \dots i-1], q[j \dots n-1], B)$;
8: **spawn** `tableauConstruction`$(p[i \dots m-1], q[0 \dots j-1], B)$;
9: **sync**;
10: `tableauConstruction`$(p[i \dots m-1], q[j \dots n-1], B)$;

case and no further division is required. Here every computation is done in a serial manner.

Algorithm 8: `tableauBaseInplace`$(p[0 \dots m-1], q[0 \dots n-1])$

Input: Array $p[0 \dots m-1]$ of size m and array $q[0 \dots n-1]$ of size n; initially $p[0 \dots m-1]$ and $q[0 \dots n-1]$ contain the input coefficients; $p[0 \dots m-1]$ and $q[0 \dots n-1]$ are overwritten during the computation.

Output: The result of right edge of Tableau is stored in array $p[0 \dots m-1]$ and down edge of the tableau is stored in array $q[0 \dots n-1]$.

```
2: for i = 0 to n − 1 do
4:     p[0] = p[0] + q[i];
6:     for j = 1 to m − 1 do
7:         p[j] = p[j] + q[j − 1];
9:     q[i] = p[m − 1];
```

Algorithm 9 `taylorShiftBase` computes triangles in base case.

Algorithm 9: `taylorShiftBase`$(p[0 \dots n-1], q[0 \dots n-1])$

Input: Array $p[0 \dots n-1]$ and array $q[0 \dots n-1]$ of size n; initially $p[0 \dots n-1]$ and $q[0 \dots n-1]$ contain the input coefficients.

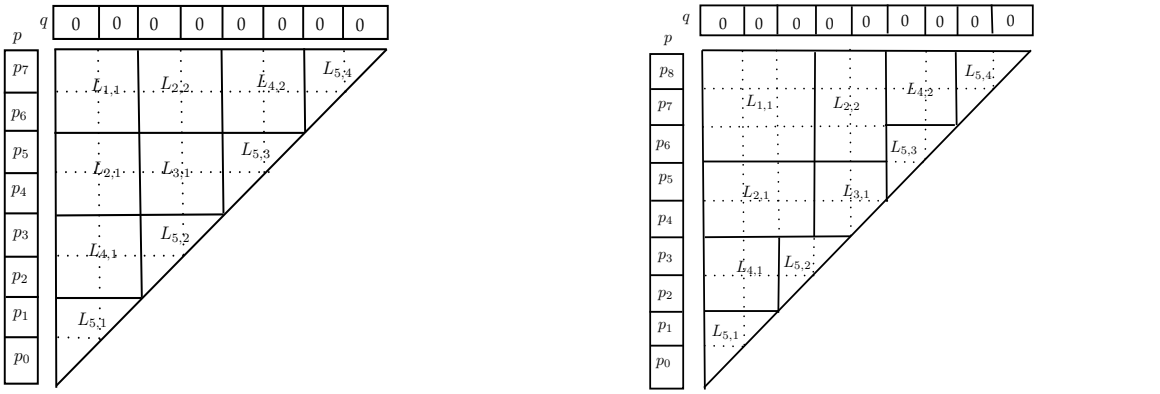
Output: Coefficient array of the polynomial $p(x+1)$ stored in the array $p[0 \dots n-1]$ such that $a[i]$ is the coefficient of the term of degree i.

```
2: for i = 0 to n − 1 do
4:     p[0] = p[0] + q[i];
6:     for j = 1 to n − i − 1 do
7:         p[j] = p[j] + q[j − 1];
```

3.1.2 Cases to consider in divide and conquer

For the example in Figure 3.2, we can see that there are two types of cases for the Divide and Conquer method, a *Regular case* (when n is power of 2) and an *Irregular case* (when n is not a power of 2). For the both regular and irregular cases, the tableau (square part) is computed first. For the square part, it is divided into four regions $L_{1,1}, L_{2,1}, L_{2,2}$ and $L_{3,1}$. Region $L_{1,1}$ will be computed first. Then region $L_{2,1}$ and region $L_{2,2}$ are computed concurrently. After that, region $L_{3,1}$ is computed. When a the square region is computed, the smaller triangles can be computed in parallel. From Figure 3.2, we see that the lower triangle is divided into 3 more regions, the tableau region $L_{4,1}$ and 2 triangle regions $L_{5,1}$ and $L_{5,2}$. The upper triangle is also

divided into 3 regions, tableau region $L_{4,2}$, and two triangles $L_{5,3}$ and $L_{5,4}$ respectively. The square regions for both triangles $L_{4,1}$ and $L_{4,2}$ can be found concurrently. Then the triangle regions $L_{5,1}, L_{5,2}, L_{5,3}, L_{5,4}$ are computed in parallel.



(a) Regular case: n is a power of 2 **(b) Irregular case: n is not a power of 2**

Figure 3.2: Case illustration for implementing Dnc strategy

3.2 Work, span, and parallelism estimates

Our work, span, and parallelism estimates for the divide and conquer approach follow the fork-join parallelism model discussed in Chapter 2.

Let $W_s(n)$ (resp. $W_T(n)$) denotes the work of the square (resp. triangle) construction and let $S_s(n)$ (resp. $S_T(n)$) denotes the span of the square (resp. triangle) construction.

The work $W_s(n)$ required for filling in the tableau satisfies

$$W_s(n) = 4W_S(n/2) + \Theta(1) \tag{3.1}$$

where, $n > 1$. Applying the *Master Theorem*[1] which implies

$$W_s(n) = \Theta(n^2) \tag{3.2}$$

for same construction the span becomes

$$S_s(n) = 3S_s(n/2) + \Theta(1) \tag{3.3}$$

[1]http://en.wikipedia.org/wiki/Master_theorem

where $n > 1$ and applying *Master Theorem* which leads to

$$S_s(n) = \Theta(n^{log_2 3}) \tag{3.4}$$

For a Pascal Triangle, the work $W_T(n)$ is

$$W_T(n) = 2W_T(n/2) + W_S(n/2) \tag{3.5}$$

Applying the *Master Theorem* yields

$$W_T(n) = \Theta(n^2) \tag{3.6}$$

And span satisfies

$$S_T(n) = S_T(n/2) + S_s(n) \tag{3.7}$$

which implies

$$S_T(n) = \Theta(n^{log_2 3}) \tag{3.8}$$

Therefore, for both constructions the parallelism is about $\Theta(n^{0.45})$. For more details see [5]

3.3 Space complexity estimate

Consider first the sequential algorithm. One can observe that, at any point of the construction of the Pascal's Triangle, the knowledge of $2n$, and only $2n$, integers (with $n = d + 1$) is needed in order to continue the construction until its completion. Moreover, the whole algorithm can be executed in place within the space allocated to the $2n$ input integers. At completion, the n output integers are located in this allocated space, most likely in its first n slots.

The same property is easy to prove for the divide and conquer approach. To do so, first establish by induction a similar property for the two-way tableau construction, see Figure 3.1. Then prove by induction the desired property for the divide-and-conquer approach of the Pascal's Triangle depicted by Figure 3.1.

Finally, we conclude that the divide and conquer approach can be executed in place using an aggregate of $2n$ integers. Observe that, in order to obtain a bit size complexity estimate, one would need to take the growth of the intermediate coefficients into account, see Chapter 5. If we assume that each input coefficient has bit size H or less, then one can verify that the whole Pascal's Triangle construction is

done within $2n(H+n)$ bits. Indeed, each output coefficient is computed from the input coefficients by n additions.

3.4 Cache complexity estimate

In their paper, Changbo Chen, Marc Moreno Maza and Yuzen Xie established the following proposition [5]. (Note that the assumption is realistic only when n and the coefficients of the input polynomial are small enough.)

Proposition 2. *Assume that each input, output or intermediate coefficient is stored in constant space. In the* ideal cache model *[10], Algorithm 7 incurs* $\Theta\left(\frac{n}{L} + n^2/(ZL)\right)$ *cache misses, which is optimal.*

Proof. Suppose, the cache size is Z words and each cache line has L words. Given an input array of length n we need a cache complexity upper bound for the two routines sketched by Figure 3.1. Let $Q_T(n)$ and $Q_S(n)$ be the number of cache misses incurred by the triangle and square routines.

Since Algorithm 7 can be run in space $2n$, we deduce that, for n small enough, the whole computation fits into cache (i.e. only cold misses occur). Therefore, there exists a real constant $\alpha > 0$ such that

$$Q_T(n) \leq \begin{cases} 2n/L + 1 & \text{if } n \leq \alpha Z \\ 2Q_T(\frac{n}{2}) + Q_S(n) & \text{otherwise,} \end{cases} \tag{3.9}$$

and

$$Q_S(n) \leq \begin{cases} 2n/L + 1 & \text{if } n \leq \alpha Z \\ 4Q_S(\frac{n}{2}) + \Theta(1) & \text{otherwise.} \end{cases} \tag{3.10}$$

We first solve Equation (3.10). Expanding this formula, we have

$$\begin{aligned} Q_s(n) &\leq 4\Big(4Q_s(\tfrac{n}{4}) + \Theta(1)\Big) + \Theta(1) \\ &\leq 4\Big(4\Big(4Q_s(\tfrac{n}{8}) + \Theta(1)\Big) + \Theta(1)\Big) + \Theta(1). \end{aligned} \tag{3.11}$$

Expanding shows that there exists an integer k such that we have

$$Q_S(n) \leq 4^k \left(\frac{2\alpha Z}{L} + 1\right) + \sum\nolimits_{j=0}^{k-1} 4^j \Theta(1), \tag{3.12}$$

where k is the smallest number of "recursive levels" necessary, such that each sub-

problem fits into cache. This number k is given by

$$k \;=\; \lceil \log_2 \left(\frac{n}{\alpha Z}\right) \rceil. \tag{3.13}$$

The $\sum_{j=0}^{k-1} 4^j \Theta(1)$ of Equation (3.17) can be simplified using the well-known identity:

$$1 + q + q^2 + \cdots + q^{e-1} = \frac{-1 + q^e}{-1 + q}.$$

So, $\sum_{j=0}^{k-1} 4^j \Theta(1)$ becomes $\Theta(4^k)$ and thus the Equation (3.17) becomes

$$\begin{aligned} Q_S(n) &\leq 4^k \left(\tfrac{2\alpha Z}{L} + 1\right) + \Theta(4^k) \\ &\leq \Theta(4^k) \left(\tfrac{2\alpha Z}{L} + 2\right) \end{aligned} \tag{3.14}$$

From Equation (3.13) we obtain the value of k and replace it in Equation (3.14), thus we have

$$\begin{aligned} Q_s(n) &\in O\left(4^{log_2(\frac{n}{\alpha Z})} \left(\tfrac{2\alpha Z}{L} + 2\right)\right) \\ &\in O\left((\tfrac{n}{\alpha Z})^2 \left(\tfrac{2\alpha Z}{L} + 2\right)\right) \\ &\in O\left(\tfrac{2n^2}{\alpha Z L} + \tfrac{2n^2}{\alpha^2 Z^2}\right). \end{aligned} \tag{3.15}$$

After simplification we obtain

$$Q_S(n) \in O\left(\frac{n^2}{ZL} + \frac{n^2}{Z^2}\right). \tag{3.16}$$

Now we solve for $Q_T(n)$ using Equation 3.9. We proceed in the same way just like Equation 3.10, after simplification we get

$$\begin{aligned} Q_T(n) &\leq 2^k Q_T\left(\tfrac{n}{2^k}\right) + \textstyle\sum_{j=0}^{k-1} 2^j Q_S\left(\tfrac{n}{2^j}\right) \\ &\leq 2^k \left(\tfrac{2\alpha Z}{L} + 1\right) + \textstyle\sum_{j=0}^{k-1} 2^j \left(\tfrac{n}{2^j}\right)^2 \left(\tfrac{1}{ZL} + \tfrac{1}{Z^2}\right) \\ &\leq \tfrac{n}{\alpha Z}\left(\tfrac{2Z}{L} + 1\right) + \textstyle\sum_{j=0}^{k-1} 2^{-j} \left(\tfrac{n^2}{ZL} + \tfrac{n^2}{Z^2}\right) \\ &\leq \tfrac{n}{\alpha Z}\left(\tfrac{2Z}{L} + 1\right) + 2\left(\tfrac{n^2}{ZL} + \tfrac{n^2}{Z^2}\right) \\ &\leq \tfrac{2n}{\alpha L} + \tfrac{n}{\alpha Z} + 2\left(\tfrac{n^2}{ZL} + \tfrac{n^2}{Z^2}\right). \end{aligned} \tag{3.17}$$

Which is actually

$$Q_s(n) \quad \in \quad O\left(\frac{n}{L} + \frac{n}{Z} + \frac{n^2}{ZL} + \frac{n^2}{Z^2}\right). \tag{3.18}$$

Using the tall cache assumption, that is, $Z \in \Omega(L^2)$, the above expression becomes

$$Q_s(n) \quad \in \quad O\left(\frac{n}{L} + \frac{n^2}{ZL}\right). \tag{3.19}$$

The optimality of this cache complexity result follows from the bound of J.W. Hong and H.T. Kung in their STOCS'81 paper "I/O complexity: the red-blue pebble game" [14]. □

Chapter 4

Blocking Taylor shift: Static approach

In this chapter, we describe a *blocking* method which is an alternate way of performing a Taylor shift computation. Section 4.1 is an informal presentation of this method. Implementation schemes with algorithms are discussed in Section 4.2 while work, span and parallelism are analyzed in Section 4.3. Space complexity and cache complexity results are provided in Section 4.4 and Section 4.5 respectively.

4.1 Blocking strategy

The divide and conquer strategy suffers a relatively low parallelism. If we recall Figure 3.1, we see that the computations in the two triangle regions do not start until the square region I is completed. But the square region parts of small triangular regions can be started before the completion of region I. Figure 4.1 gives an idea of computing a Pascal Triangle by a blocking method instead of proceeding with a divide and conquer strategy.

For clarity, let us recall Figure 3.2 and observe that region $L_{3,1}$ (which located in the bottom-right corner of the tableau) is computed before region $L_{4,1}$ and $L_{4,2}$ (which are the square regions of upper and lower triangles). But here we can compute all three regions concurrently since these three regions become essentially available for computing in parallel at the same time. To be benefited, a new approach is introduced called the *blocking method*.

In their paper [6] Changbo Chen, Marc Moreno Maza and Yuzhen Xie discuss a *blocking strategy*, where the entire Pascal Triangle is partitioned into square blocks, all of the same format, say $B \times B$. Those blocks are traversed one *band* after another,

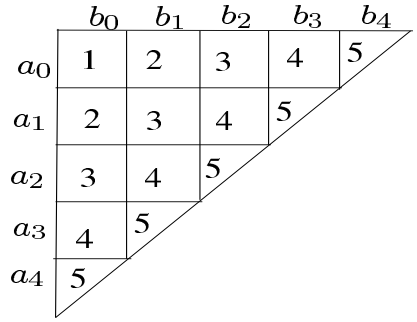


Figure 4.1: Blocking Strategy in Pascal Triangle

where a band consists of blocks on the same segment perpendicular to the principal diagonal. On Figure 4.1, "1", "$2, 2$", "$3, 3, 3$" form consecutive blocks. We refer to this strategy as the *static blocking strategy.*

4.2 Implementation scheme for the static blocking strategy

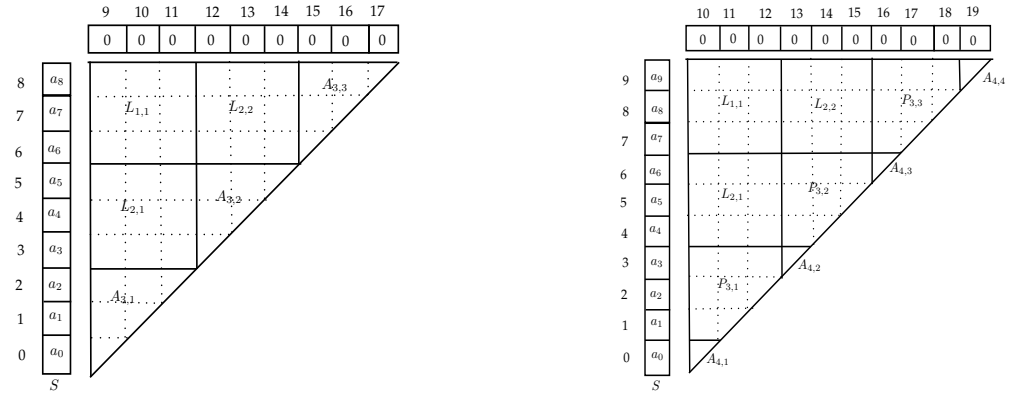
As in Section 2.5, let d be the degree of the input polynomial and define $n = d+1$. Two cases have to considered: the block order B divides n, that we call the *regular case* and B does not divide n, that we call the *irregular case.* The example in Figure 4.2(a), where $n = 9$ and $B = 3$, illustrates the regular case. Here the computation can be done in three steps. First step: tableau $L_{1,1}$ is computed. Second step: two tableau $L_{2,1}$ and $L_{2,2}$ are computed in parallel. Lastly: three small Pascal Triangles $L_{3,1}$, $L_{3,2}$, $L_{3,3}$ are computed concurrently.

Regarding the irregular case we consider $n = 10$ and $B = 3$, see Figure 4.2(b). After two diagonal rows of the blocks from the top left corner of triangle, a special case occurs, we call it the *polygonCase*. From Figure 4.2 we see that three polygons $P_{3,1}$, $P_{3,2}$, $P_{3,3}$ (we call these special shapes *polygon* as their shape neither looks like a 'Tableau' nor a 'Triangle') occur after the square blocks are done. In the last step, four Pascal Triangles are computed concurrently.

Figure 4.2 shows both cases for the blocking strategy.

4.2.1 Regular case

When the degree of a polynomial n is divisible by B, the regular case Algorithm 10 will work. Here, Line 6 to Line 12 computes the tableau regions in parallel. Once the

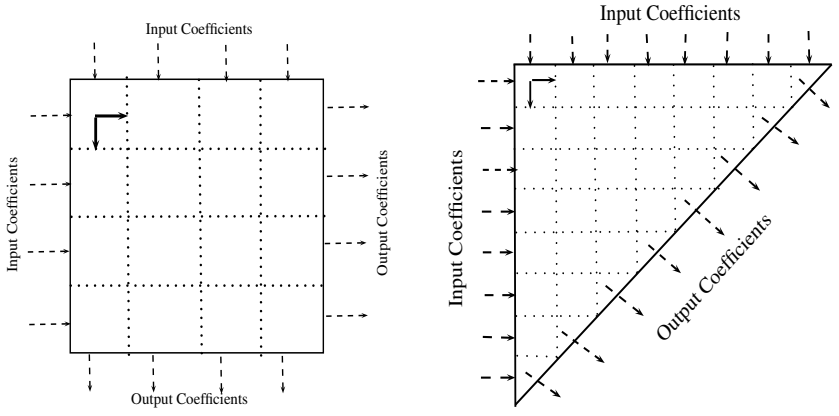


(a) Regular case: B divides n **(b) Irregular case: B does not divide n**

Figure 4.2: Case illustration for implementing the blocking strategy

tableau part is done, Line 13 to Line 16 is calculates the triangle regions concurrently. All computations here are done in $2n$ space. (Thus the results are overwritten on input space.)

Figure 4.3 shows both the tableau and the triangle regions. For the both regions, each of the smaller blocks inside them is computed by adding inputs from left and top edge of each block. After addition each of the smaller blocks containing result becomes input of their right side and downside blocks. Finally, when all these smaller blocks are computed, the outputs for the tableau block become available on its right edge and down edge (for triangle it is on its diagonal edge).



(a) Tableau block **(b) Triangle block**

Figure 4.3: Tableau and triangle block

Algorithm 10: `staticBlockingRegular`$(a[0 \ldots n-1], n, B)$

Input: $a[0 \ldots n-1]$ is the coefficient array (dense representation) of a univariate polynomial $p(x)$ of degree d where $n = d + 1$ ($a[i]$ is the coefficient of the term of degree i); B is the base size (threshold).

Output: Coefficient array of the polynomial $p(x+1)$ stored in the array $a[0 \ldots n-1]$ such that $a[i]$ is the coefficient of the term of degree i.

```
1:  m = 2n;
2:  b[0, ..., n − 1] = a[0, ..., n − 1];
3:  b[n, ..., m − 1] = 0;
4:  k = n/B;
5:  if k > 1 then
6:      tableauBaseBlock(b[n − B ... n], B);
7:      for i = 2 to k − 1 do
8:          for j = 0 to i − 2 do
9:              t = n + (2j − i)B;
10:             spawn tableauBaseBlock(b[t ... t + B], B);
11:         spawn tableauBaseBlock(b[n + (i − 2)B ... n + iB − B], B);
12:         sync;
13: for i = 0 to k − 2 do
14:     spawn taylorBaseBlock(b[2iB ... 2iB + B], B);
15: spawn taylorBaseBlock(b[2(k − 1)B ... 2kB − B], B);
16: sync;
    /* copy results from b to a                                          */
17: for i = 0 to k − 1 do
18:     for j = 0 to B − 1 do
19:         a[j + iB] = b[j + 2iB];
```

The pseudocode for computing the tableau regions is shown in Algorithm 11. In this algorithm, Line 2 and 3 compute the first row of the tableau region. Line 4 to Line 7 computes the middle rows. Finally, last rows of the tableau are computed at line 8 to Line 10. Results of the tableau are overwritten on input array, thus, the whole computation is done in $2n$ space.

Algorithm 11: `tableauBaseBlock`$(a[0,\ldots,n-1],n)$

Input: $a[0\ldots n-1]$ is an array of size n containing input of left and top edge of each tableau, shown in Figure 4.3 (a).

Output: Array $a[0\ldots n-1]$ overwritten by the outputs which act as input for right side and downside tableau blocks, see Figure 4.3 (a).

```
1:  m = 2n;
    /* first row                                                    */
2:  for i = n to m − 1 do
3:      a[i] = a[i] + a[i − 1];
    /* middle rows                                                  */
4:  for i = n − 2 to 1 do
5:      for j = 0 to n − 1 do
6:          a[i + j + 1] = a[i + j] + a[i + j + 2];
7:      i = i − 1;
    /* first element of last row                                    */
8:  a[0] = a[0] + a[2];
    /* rest element of last row                                     */
9:  for i = 1 to n − 1 do
10:     a[i] = a[i − 1] + a[i + 2];
    /* first element of first row                                   */
11: a[n] = a[n − 1];
```

Algorithm 12, `taylorBaseBlock` is showing the procedure for triangle block computation. The picture for a single triangle block is shown in Figure 4.3 (b). Just like tableau block, here also all the computations are done in place, within an aggregate of $2n$ integer coefficients.

Algorithm 12: `taylorBaseBlock`$(a[0,\ldots,n-1],n)$

Input: $a[0\ldots n-1]$ is an array of size n containing input of left and top edge of each triangle, shown in Figure 4.3 (b).

Output: Array $a[0\ldots n-1]$ overwritten by the outputs. Outputs become available on the diagonal edge of each triangle, see Figure 4.3 (b).

```
2: for i = 0 to n − 1 do
4:     a[n − 1] = a[n − 1] + a[n + i];
6:     for j = n − 2 to i do
7:         a[j] = a[j] + a[j + 1];
8:         j = j − 1;
```

4.2.2 Irregular case

Algorithm 13 works for static blocks for irregular cases (n is not divisible by B). In this algorithm, Line 8 to Line 15 computes the tableau regions in parallel. When the tableau part is done, Line 16 to Line 20 calculates the special regions which we have named *Polygon* regions. At the end, Line 21 to Line 24 computes triangle regions concurrently. Every computation is done here in place so the result overwrites the input array.

Algorithm 13: `staticBlockingIrregular`$(a[0 \dots n-1], n, B)$

Input: $a[0 \dots n-1]$ is the coefficient array (dense representation) of a univariate polynomial $p(x)$ of degree d where $n = d + 1$ ($a[i]$ is the coefficient of the term of degree i); B is the base size (threshold).

Output: Coefficient array of the polynomial $p(x+1)$ stored in the array $a[0 \dots n-1]$ such that $a[i]$ is the coefficient of the term of degree i.

```
1:  r = n mod B;
2:  q = n/B;
3:  m = 2n;
4:  k = B − r − 1 ;
5:  k = (k > 0)?k : 1;
6:  b[0, . . . , n − 1] = a[0, . . . , n − 1];
7:  b[n, . . . , m − 1] = 0;
8:  if q > 1 then
9:      tableauBaseBlock(b[n − B . . . n], B);
10:     for i = 2 to q − 1 do
11:         for j = 0 to i − 2 do
12:             t1 = n + (2j − i)B;
13:             spawn tableauBaseBlock(b[t1 . . . t1 + B], B);
14:         spawn tableauBaseBlock(b[n + (i − 2)B . . . n + iB − B], B);
15:         sync;
16: for i = 0 to q − 2 do
17:     t2 = n + (2i − q)B;
18:     spawn polygonBase(b[t2 . . . t2 + B], B, r, k);
19: spawn polygonBase(b[n + (q − 2)B . . . n + qB − B], B, r, k);
20: sync;
21: for i = 0 to q − 1 do
22:     spawn taylorBaseBlock(b[2iB . . . 2iB + r], r);
23: spawn taylorBaseBlock(b[2qB . . . 2qB + r], r);
24: sync;
25: Copy results from b[0 . . . n − 1] to a[0 . . . n − 1];
26: return a[0 . . . n − 1] ;
```

In Algorithm 14 describes the polygon construction. The special shape, which is neither a tableau, nor a triangle is handled by this algorithm. This type of special

shape, which looks more like combination of “Rectangle” and “Trapezoid”, can be divided into two areas. One is a rectangular area at the top of the polygon and the other is trapezoidal area. Figure 4.4 shows the picture of a polygon block. Each of the smaller blocks inside the polygon is computed in the same way like tableau blocks. Finally, outputs become available on right edge, down edge and on the diagonal edge of the polygon.

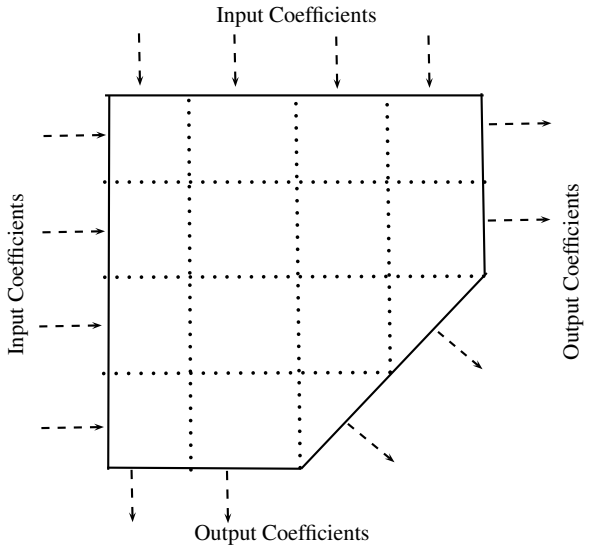


Figure 4.4: Polygon block.

In Algorithm 14, Line 2 to Line 7 calculates the rectangle rows or the top rows of the polygon. Rows with trapezoidal shape are computed from Line 8 to Line 12. The last row of polygon is computed at Line 13 to Line 15. Here results also overwrite the input array, thus doing calculation in place.

Algorithm 14: `polygonBase`$(a[0,\ldots,n-1], n, r, k)$

Input: $a[0\ldots n-1]$ is an array of size n containing input of left and top edge of each smaller block inside polygon, shown in Figure 14; r is the remainder of (n mod B), k is the number of trapezoidal rows in polygon.

Output: Array $a[0\ldots n-1]$ overwritten by the outputs. Outputs become available on the right edge, down edge and diagonal edge of each polygon, see Figure 4.4.

```
1:  m = 2n;
    /* first row                                                    */
2:  for i = n to m − 1 do
3:      a[i] = a[i] + a[i − 1];
    /* rectangle rows                                               */
4:  for i = n − 2 to k do
5:      for j = 0 to n − 1 do
6:          a[i + j + 1] = a[i + j] + a[i + j + 2];
7:      i = i − 1;
    /* trapezoid starts                                             */
8:  for i = k − 1 to 1 do
9:      a[k + 1] = a[k + 1] + a[i];
10:     for j = k + 2 to n + i do
11:         a[j] = a[j] + a[j − 1];
12:     i = i − 1 ;
    /* first element of last row                                    */
13: a[0] = a[0] + a[k + 1];
    /* rest element of last row                                     */
14: for i = 1 to r do
15:     a[i] = a[i − 1] + a[i + k + 1];
```

4.3 Work, span, and parallelism estimates

Let B be the order of a block. Then the work for each block is $\Theta(B^2)$. If the number of elements is n, then there will be n/B "parallel steps" or we can say n/B *bands* (or diagonal rows) from the upper left corner of a triangle. Each of these bands have at

most n/B blocks. Thus, we have $\Theta(n/B)^2$ block. Hence the work for computing the Pascal Triangle is:

$$\begin{aligned} W_B(n) &\in \Theta(B^2) \times \Theta((n/B)^2) \\ &\in \Theta(n^2). \end{aligned} \tag{4.1}$$

As the span for each block is $\Theta(B^2)$, the span for the whole algorithm is:

$$\begin{aligned} S_B(n) &\in \Theta(B^2) \times n/B \\ &\in \Theta(Bn). \end{aligned} \tag{4.2}$$

The resulting parallelism is $\Theta(n^2/(Bn))$, which is $\Theta(n/B)$.

4.4 Space complexity estimate

For a $B \times B$ block, the computation is sequential and is done in place within $2B$ integers. At the k-th parallel step there are k blocks which requires $2kB$ integers in total. Therefore, the whole algorithm can be run with an aggregate of $2n$ integers (letting $k = n/B$). We observe that this analysis does not take into account the growth of the intermediate coefficients. See Section 3.3 and Chapter 5 for more details on this.

4.5 Cache complexity estimate

Let α be the constant introduced in the cache complexity analysis of the divide and conquer approach. Assume that $B = \alpha Z$. Then, the number of cache misses for each block is $2B/L + 1$ and the total number of cache misses is

$$Q(n) = \Theta((n/B)^2(2B/L + 1)) = \Theta(n^2/(BL)) = \Theta(n^2/(ZL)).$$

Therefore, provided that $B = \alpha Z$, we retrieve the optimal cache complexity result established for the divide and conquer approach.

Chapter 5

Analysis of Workload in the Case of Integer Coefficients

In the fork-join parallelism model, the work of each strand is assumed to have an unit cost. Under this assumption, the analyses that we conducted in the previous chapters show that the work for constructing Pascal's Triangle is quadratic in order n. As long as all necessary additions can be performed correctly with machine integer arithmetic, this assumption is realistic. However, for n large enough, software integer arithmetic is required and this assumption is no longer acceptable.

More generally, this phenomenon invalidates our complexity analyses of the divide and conquer approach, and the blocking strategy for Taylor shift computations. To understand this, let us consider the divide and conquer approach for the Taylor shift, as described in Section 3.2. We revisit Figure 3.1 below.

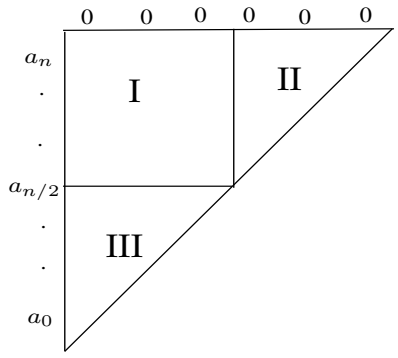


Figure 5.1: Pascal Triangle.

Under the assumption that each strand has unit work, Regions II and III in Figure 5.1 have the same work and span. Suppose now that we are using a machine

of 2-bit machine word. Then Region II requires more work (in terms of machine word operations) than Region III.

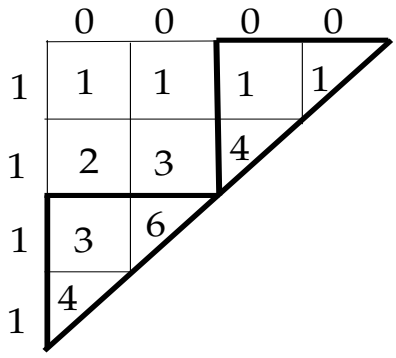


Figure 5.2: Example figure of DnC in Pascal Triangle

Suppose from now on that we are using a machine of b-bit machine word, for an arbitrary integer $b > 1$. Consider two positive integers A and B in radix b representation

$$A = \sum_{i=0}^{s-1} A_i b^i \quad \text{and} \quad B = \sum_{i=0}^{t-1} B_i b^i,$$

with $A_{s-1} \neq 0, 0 \leq A_i < b$ and $B_{t-1} \neq 0, 0 \leq B_i < b$. The integers s and t are called the sizes of A and B, denoted (respectively) by $\#(A)$ and $\#(B)$. We clearly have

$$\max(\#(A), \#(B)) \leq \#(A + B) \leq \max(\#(A), \#(B)) + 1.$$

If A and B are random (with an uniform distribution), we have $\#(A + B) = \max(\#(A), \#(B)) + 1$ with probability $1/2$. This explains the growth of the coefficients in the Pascal Triangle and thus, for n large enough, the unbalanced work between the two recursive calls in the divide and conquer construction of the Pascal Triangle.

The next two sections are dedicated to an analysis of this phenomenon and its impact on the work and span of parallel Taylor shift computations by means of the blocking strategy. We focus, indeed, on this scheme since its parallelism is higher than that of the divide and conquer strategy. In fact, the goal of this section is to obtain complexity estimates that will support an improved implementation of this scheme, reported in Section 7.

In Section 5.3, we analyze the parallelization overheads of the blocking strategy, which, in our opinion, bring some interesting and unexpected results. Finally, in Section 5.4, we turn our attention to locality issues. We establish formulas for choosing

an initial block order B, and, if appropriate, changing this block order during the computations, so as to minimize cache misses.

5.1 Work for the blocking strategy

We consider a triangular grid as in Figure 3.1, where the top left corner is the origin, with Cartesian coordinates $(0, 0)$. The coefficient in the grid with coordinates (k, ℓ) (where k is the row index and ℓ is the column index) is denoted by $c_{k,\ell}$. The grid has n rows and n columns.

Initially, the coefficients satisfying either $k = 0$ or $\ell = 0$ are known and called the *initial coefficients*. The other coefficients are given by

$$c_{k,\ell} = c_{k-1,\ell} + c_{k,\ell-1}.$$

We denote by H the maximum size of the absolute value of an initial coefficient. We denote by $s_{k,\ell}$ the size of the absolute value of $c_{k,\ell}$. For $k > 0$ and $\ell > 0$, we clearly have

$$s_{k,\ell} \leq H + k + \ell - 1. \tag{5.1}$$

Since this upper bound of $s_{k,\ell}$ is quite pessimistic, we introduce a constant $p \in [0, 1]$ such that $S_{k,\ell}$ (below) is expectedly a sharper upper bound of $s_{k,\ell}$

$$S_{k,\ell} = H + p(k + \ell - 1). \tag{5.2}$$

For instance if all initial coefficients are positive and have size H, we can take $p = 1/2$ and $S_{k,\ell}$ is actually the expected value of $s_{k,\ell}$. As we shall see, this parameter p has little impact on the key results. However, not using this parameter through our calculations would imply that all our other estimates would rely on quite pessimistic estimates, which would be questionable.

Recall that we are interested in the blocking strategy. Let $B > 1$ be the block order, we assume that B divides n. Consider a square $B \times B$ block whose top left corner has coordinates (k, ℓ). We assume that all $c_{k,\ell+j}$ for $j = 0 \cdots B$ and all $c_{k+i,\ell}$ for $i = 0 \cdots B$ are known. The work $C_{B,k,\ell}$ for computing all the other coefficients of the block is given by

$$C_{B,k,\ell} = \sum_{i=1}^{B} \sum_{j=1}^{B} S_{k+i,\ell+j}. \tag{5.3}$$

Indeed the work required for adding two numbers is in the order of the size of their

sum. Then, elementary calculations yield

$$C_{B,k,\ell} = B^2(H + p(k + \ell + p)). \tag{5.4}$$

Similarly, we obtain the work $T_{B,k,\ell}$ of a triangular $B \times B$ block whose top left corner has coordinates (k, ℓ):

$$T_{B,k,\ell} = \sum_{i=1}^{B} \sum_{j=1}^{B+1-i} S_{k+i,\ell+j} = \frac{1}{6} B(B+1)(2Bp + p + 3pk + 3H + 3p\ell). \tag{5.5}$$

We are now ready to compute the work $W(n)$ required for calculating all coefficients (except the initial ones) in the Pascal Triangle.

Proposition 3. *We have*

$$W(n) = \frac{1}{6} n(n+1)(3\,H + 2\,p\,n + p). \tag{5.6}$$

Proof ▷ We first observe that two square (resp. triangular) $B \times B$ blocks with top left corner (k, ℓ) and (k', ℓ') have the same work if and only if they are on the same band, that is, whenever $k + \ell = k' + \ell'$ holds. Secondly, we observe that the j-th band of square $B \times B$ blocks, for $0 \leq j \leq n/B - 2$, has $j + 1$ blocks. Thirdly, the band of triangular $B \times B$ blocks has n/B blocks. This leads to

$$W(n) = \sum_{j=0}^{n/B-2} (j+1)\, C_{B,j\,B,0} + \frac{n}{B} T_{B,(n/B-1)B,0}. \tag{5.7}$$

Elementary calculations lead to the conclusion. ◁

The formula of Proposition 3 shows that the work is cubic in n. It is also independent of B, which was naturally expected. When $p = 0$, we retrieve the quadratic case of Section 4.

5.2 Span for the blocking strategy

We are now interested in the span $S(n)$ required for calculating all coefficients (except the initial ones) by means of the blocking strategy.

Proposition 4. *We have*

$$S(n) = \frac{1}{6} B(6Hn - 3HB + 3pn^2 - pB^2 + p + 3H + 3pn). \tag{5.8}$$

PROOF ▷ The span of each band (of square or triangular $B \times B$ blocks) is equal to the work of a block on that band. Indeed, each block is computed serially. Thus, following the proof of Proposition 3, we have

$$S(n) = \sum_{j=0}^{n/B-2} C_{B,j\,B,0} + T_{B,(n/B-1)B,0}. \tag{5.9}$$

Elementary calculations lead to the conclusion. ◁

The formula of Proposition 4 shows that the span is quadratic in n and cubic in B. When $p = 0$, we have $S(n) = \frac{BH}{2}(2n - B + 1)$ and thus, in this case, the parallelism is

$$\frac{n(n+1)}{B(2n - B + 1)}, \tag{5.10}$$

which is asymptotically equal[1] to $\frac{n}{2B}$ for a fixed B.

When $p = 1$, we have

$$S(n) = \frac{B}{6}(6Hn - 3HB + 3n^2 - B^2 + 3H + 3n + 1), \tag{5.11}$$

and thus, in this case, the parallelism is

$$\frac{n(n+1)(3\,H + 2\,p\,n + p)}{B(6Hn - 3HB + 3n^2 - B^2 + 3H + 3n + 1)}. \tag{5.12}$$

which is asymptotically equal $\frac{2n^3}{3Bn^2} = \frac{2n}{B}$, for a fixed B and a fixed H. Finally, when $p = 1/2$, similar calculations yield a parallelism which is asymptotically equal to $\frac{2n}{3B}$.

The above results show that taking into account the growth of the coefficients in Pascal's Triangle construction (or in Taylor shift computation) by the blocking strategy increases the parallelism by a factor which is at most 4.

5.3 Estimating parallelization overheads

In this section, we estimate the burdened span $S_b(n)$ incurred for calculating all coefficients (except the initial ones) by means of the blocking strategy. Recall that, in the fork-join parallelism model (thus assuming that all strands run in unit time) the burdened span is the maximum number of continuations (thus `cilk_spawn` statements) along a critical path. The instruction stream DAG of the blocking strategy consists of n/B binary tress $T_0, T_1, \ldots, T_{n/B-1}$ such that

[1]For the notion of *asymptotically equal*, see `http://en.wikipedia.org/wiki/Big_O_notation`.

- T_i is the instruction stream DAG of the `cilk_for` loop executing the i-th band, and
- each leaf of T_i is connected by an edge to the root of T_{i+1}.

Consequently, we have

$$\begin{aligned} S_b(n) &= \sum_{i=1}^{n/B} \log(i) \\ &= \log(\prod_{i=1}^{n/B} i) \\ &= \log(\Gamma\left(\tfrac{n}{B} + 1\right)). \end{aligned} \tag{5.13}$$

Using Stirling's Formula, we deduce the following.

Proposition 5. *We have*

$$S_b(n) \in \Theta\left(\frac{n}{B}\log(\frac{n}{B})\right). \tag{5.14}$$

This result implies that the burdened parallelism (that is, the ratio work to burdened span) is sublinear. Another way to interpret this negative result is as follows. When B is replaced by its half, the (non-burdened) span is essentially multiplied by 2 while the burdened span is multiplied by a factor greater than 2. In other words, considering the (non-burdened) span only, the replacement of B by its half seems like a good idea while it is a bad idea from the burdened span point of view.

5.4 Choosing the block order

We turn our attention to locality issues. More precisely, for an ideal cache of size Z and cache-line L, we ask how to choose the block order B so as to minimize cache misses.

We first determine the space requirement $R(B, k, \ell)$ for computing a square $B \times B$ block whose top left corner has coordinates (k, ℓ). As before, we assume that all $c_{k,\ell+j}$ for $j = 0 \cdots B$ and all $c_{k+i,\ell}$ for $i = 0 \cdots B$ are known. We assume that the largest coefficients in that block will be at $c_{k+B,\ell+j}$ for $j = 0 \cdots B$ and at $c_{k+i,\ell+B}$ for $i = 0 \cdots B$. Those coefficients are actually the result of computing our block rooted at (k, ℓ). Indeed, we assume that the space used for the other coefficients of the block is recycled for storing the output coefficients. Thus we have

$$R(B, k, \ell) = \sum_{i=0}^{B-1} S_{k+i,\ell+B} + \sum_{j=0}^{B} S_{k+B,\ell+j}. \tag{5.15}$$

Then, elementary calculations yield

$$R(B, k, \ell) = 2HB + H + 2pBk + pk + 3pB^2 - pB + 2pB\ell + p\ell - p. \tag{5.16}$$

In particular, we have

$$R(B, 0, 0) = 2HB + H + 3pB^2 - pB - p. \tag{5.17}$$

Since each block is processed by one thread, and assuming that each thread has a private (Z, L)-ideal cache, we should always have

$$R(B, k, \ell) \leq Z. \tag{5.18}$$

Experience shows that $R(B, k, \ell)$ should a portion α of Z. In our computations, $\alpha = 1/4$ yields the best results. Given H, p, α, Z one can solve for B the equation $R(B, 0, 0) = \alpha Z$. We obtain

$$B = \frac{p - 2H + \sqrt{13p^2 - 16Hp + 4H^2 + 12p\alpha Z}}{6p}. \tag{5.19}$$

Fixing $\alpha = 1/4$, $H = 1000$, $Z = 32Kb$, which are realistic values, and letting p be successively 0, 1/2 and 1, we obtain $B = 30.85$, $B = 31.53$, $B = 31.89$. For the α, H, Z, Figure 5.3 plots the block order B as a function of the probability p. We observe that p has a little impact on the value of B.

From now on we assume $\alpha = 1/4$. Formula (5.19) tells us how to choose the initial block order. However, due to the potential growth of the coefficients in the successive bands, we consider replacing B by its half after s steps. The number s is such that $R(B, s, 0) = Z$, that is, after s rows, we can no longer compute a square $B \times B$ block without incurring cache misses other than cold misses.

We solve the system of equations and inequalities

$$\begin{cases} R(B, 0, 0) = \alpha Z \\ R(B, s, 0) = Z \\ B > 1, H > 1, Z > 0, s > 0, 1 > p > 0. \end{cases} \tag{5.20}$$

Using the `RealTriangularize` command of the `RegularChains` library in MAPLE

we obtain the following defining expressions for B and s, as functions of H, Z, p:

$$\begin{cases} (2Bp+p)s + H - p - Z + (2H-p)B + 3pB^2 = 0 \\ 12pB^2 + (-4p+8H)B + 4H - 4p - Z = 0. \end{cases} \tag{5.21}$$

We take advantage of these estimates in the algorithms of the next chapters. Fixing $\alpha = 1/4$, $H = 1000$, $Z = 32Kb$, Figure 5.4 plots the row number s as a function of the probability p. We observe that p has a large impact on the value of s.

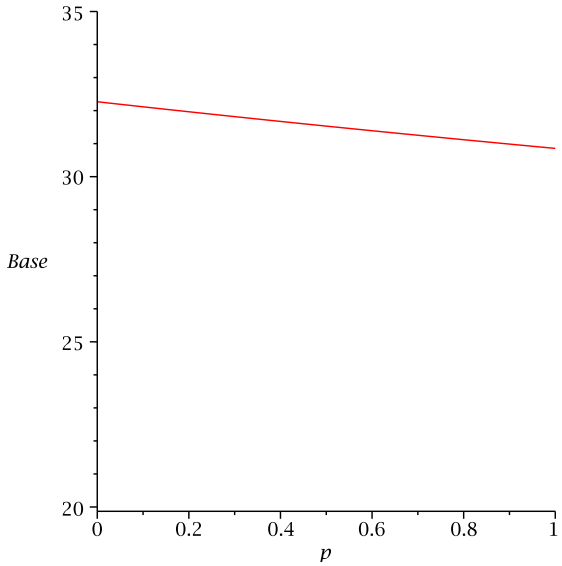


Figure 5.3: B as a function of p.

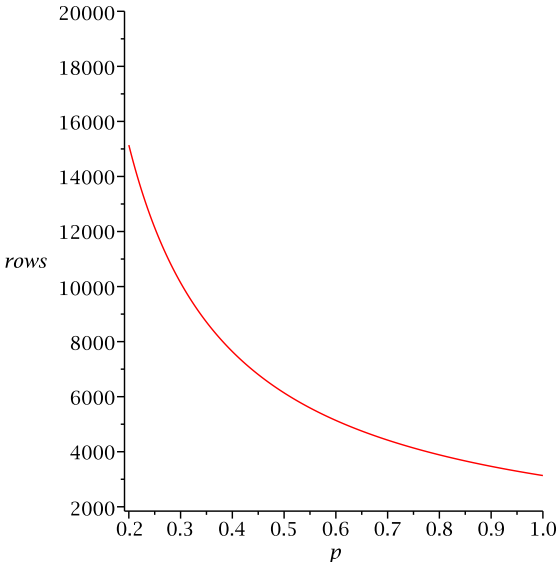


Figure 5.4: s as a function of p.

Chapter 6

Divide and Conquer Taylor shift: Dynamic approach

This chapter describes implementation techniques for Taylor shift computations using the divide and conquer scheme. Based on the study conducted in Section 5, we know that two recursive calls in the scheme are likely not to have the same work load due to the growth of the coefficients. In Chapter 3, we were using a fixed base case in the divide and conquer scheme, thus not taking into account that the work load can be different for two square (or triangular) regions of the same size. Two techniques to overcome this limitation are discussed in this chapter.

A first approach, presented in Section 6.1, is to *dynamically* determine the base case, that is, the order of a square (resp. triangular) region below which the recursive division stops. A second approach, proposed in Section 6.2 is to compute the triangular regions with heavy work load as the "sum" if two triangular regions with half of the work load each. We named this second technique *partial sum*. We also consider the combination of these two techniques, that we call *combo*. Experimental results based on these techniques are reported in Section 6.3.

6.1 Divide and conquer scheme with dynamic base case

Let $p(x) \in \mathbb{Q}[x]$ be a polynomial of degree d (where $n = d + 1$) whose coefficients are stored in an array $p[0 \dots n-1]$, where $a[i]$ is the coefficient of the term of degree i. Let $q[0 \dots n-1]$ be another array of length n, where all coefficients are zero. Let α be a portion of the L1 cache, typically equal to $1/4$, for the reasons noted in Section 5.4.

Algorithm 15: `taylorShiftGeneralDynamic`$(p[0\ldots n-1], q[0\ldots n-1], \alpha)$

Input: $p[0\ldots n-1]$ is the coefficient array (dense representation) of an univariate polynomial $p(x)$ of degree d where $n = d+1$ ($a[i]$ is the coefficient of the term of degree i); $q[0\ldots n-1]$ is another array of length n, where all coefficients are initially zero; both $p[0\ldots n-1]$ and $q[0\ldots n-1]$ are overwritten; α is a portion of the L1 cache, typically equal to 1/4.

Output: Coefficient array of the polynomial $p(x+1)$ stored in the array $p[0\ldots n-1]$ such that $a[i]$ is the coefficient of the term of degree i.

```
1:  Randomly choose 10 coefficients from p;
2:  Randomly choose 10 coefficients from q;
3:  Find the largest coefficient among them;
4:  H = Bit size of largest coefficient;
5:  B = 1/2 − H/3 + (−3+4H^2+3αZ)^{1/2} / 6
6:  if B ≤ B_min then
7:  |  B = B_min
8:  ;
9:  if n ≤ B then
10: |  taylorShiftBase(p[0...n−1], q[0...n−1]);
11: else
12: |  m = (n+1)/2;
13: |  tableauConstruction(p[0...m−1], q[0...m−1], B);
14: |  spawn taylorShiftGeneralDynamic(p[0...m−1], q[m...n−1], α);
15: |  spawn taylorShiftGeneralDynamic(p[m...n−1], q[0...m−1], α);
16: |  sync;
```

Algorithm 15 applied to (p, q, α) computes and stores the coefficients of $p(x+1)$ in the array $p[0\ldots n-1]$. Algorithm 15 is based on the divide and conquer scheme discussed in Chapter 3. However, and to the contrary of Algorithm 6, the base case (or threshold) B is determined dynamically.

The idea is to use Formula (5.19) from Chapter 5 in order to decide whether the current square (or triangular) region should be divided or not. When B is too small, we know from Formula (5.14) that the burdened span can be significantly larger than the span, thus reducing performances. We have determined experimentally that B should not be smaller than 25. This value should not be seen as a "Voodoo parameter". It is directly imposed by the parallelization overheads (number of cycles for `cilk_span`, etc.), see Sections 2.8.3 and 2.8.5. In our algorithms we refer to this as B_{min}.

In the context of Algorithm 15, we use Formula (5.19) as follows: We do not

compute the maximum absolute value H of an initial coefficient, that is, the maximum absolute value of a coefficient among those of p and q. Indeed, this would increase the work and the span of Algorithm 15 in a non-acceptable way. (Think about the recursive calls!) For this reason, we use a statistical approach: we pick randomly 10 coefficients in p and 10 coefficients in q, then use them to estimate H.

6.2 Dynamic divide and conquer scheme with partial sum

As in Chapter 5, we view the Pascal Triangle as a triangular grid, where the top left corner is the origin, with Cartesian coordinates $(0, 0)$. The coefficient in the grid with coordinates (k, ℓ) (where k is the row index and ℓ is the column index) is denoted by $P(k, \ell)$ The grid has n rows and n columns.

Computing all the coefficients $P(k, \ell)$ in a parallel fashion can be done in different ways, in particular using a divide and conquer scheme. Let us recall this scheme. Suppose that n is even and let $q = n/2$. We partition the Pascal Triangle P in three regions that we denote by C, P^+, and P^- and that are defined as follows:

- for $0 \leq k, \ell \leq q - 1$ we have $C(k, \ell) = P(k, \ell)$ (which is a square region, often called a *tableau*),
- for $q \leq \ell \leq n-1$ and $k+\ell \leq n-1$, we have $P^+(k, \ell) = P(k, \ell)$ (this is top-right triangle),
- for $q \leq k \leq n-1$ and $k+\ell \leq n-1$, we have $P^-(k, \ell) = P(k, \ell)$ (this is down-left triangle).

Observe that computing the elements of P^+ and P^- require the knowledge of those of C. However, one can compute the coefficients of P^- in three stages:

1. Assuming that $P(q - 1, \ell) = 0$ for $0 \leq \ell \leq n - q$, compute P^- and call P_0^- the result,
2. Assuming that $P(q - 1, \ell)$ is known for $0 \leq \ell \leq n - q$ and setting $c_k = 0$ for $q \leq k \leq n - 1$, compute P^- and call P_1^- the result,
3. add P_0^- and P_1^- element-wise to deduce P^-.

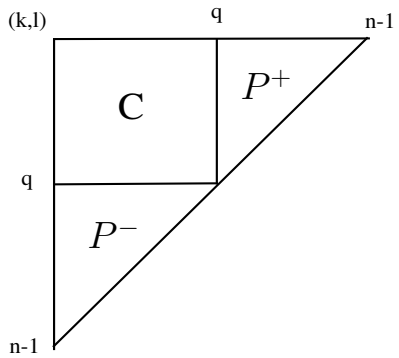


Figure 6.1: Pascal Triangle.

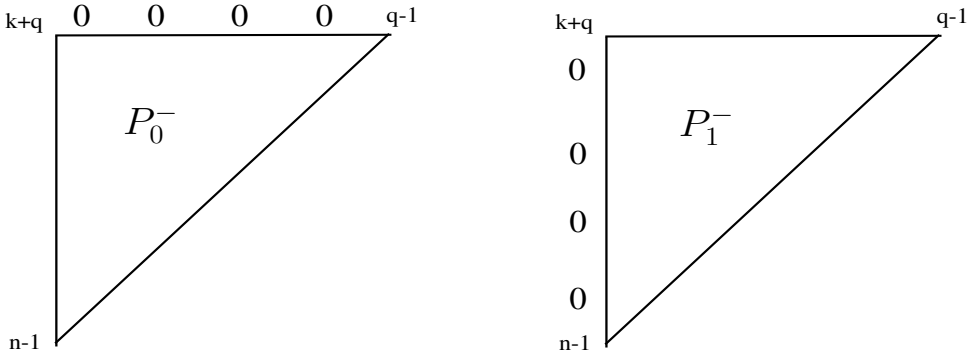


Figure 6.2: Lower triangle P^- computed in two steps: P_0^- and P_1^-.

We call this approach the *partial sum trick*. Figure 6.1 and Figure 6.2 show how the partial sum trick is applied to a down-left triangle.

Let us denote by $W_P, W_C, W_{P^-}, W_{P^+}, W_{P_0^-}$, and $W_{P_1^-}$ the work (bit operations) for computing P, C, P^+, P^-, P_0^-, and P_1^-. Similarly, let us denote by $S_P, S_C, S_{P^-}, S_{P^+}, S_{P_0^-}$, and $S_{P_1^-}$ the span (bit operations) for computing P, C, P^+, P^-, P_0^-, and P_1^-.

If the partial sum trick is not applied, then the parallelism of the divide and conquer is given by:

$$T_1/T_\infty = \frac{W_C + W_{P^-} + W_{P^+}}{S_C + \max(S_{P^+}, S_{P^-})}. \tag{6.1}$$

If the partial sum trick is applied, then the parallelism becomes:

$$T_1/T'_\infty = \frac{W_C + W_{P^-} + W_{P^+}}{\max(S_C, S_{P_0^-}) + \max(S_{P^+}, S_{P_1^-})}. \tag{6.2}$$

In the above, we are neglecting the work overhead and the span overhead in the third

stage of the partial sum trick. This is because, in practice, we only care about the *diagonal elements* of P, that is, the coefficients $P^-(k,\ell)$ for $\ell + k = n - 1$.

In order to further compare the parallelism of the two variants, we make two following genericity assumptions. We assume that we have:

- $S_{P_1^-} \geq S_{P^+}$ and
- $S_C \geq S_{P_0^-}$.

For instance, these assumptions hold if all c_i's are equal. Under these assumptions, we have

$$\frac{T_1/T_\infty}{T_1/T'_\infty} = \frac{\max(S_C, S_{P_0^-}) + \max(S_{P^+}, S_{P_1^-})}{S_C + \max(S_{P^+}, S_{P^-})} = \frac{S_C + S_{P_1^-}}{S_C + S_{P^-}}. \tag{6.3}$$

Therefore, for the partial sum trick to improve the parallelism of the original divide and conquer approach we need S_C to be relatively small compared to $S_{P_1^-}$, which itself should be small compared to S_{P^-}. The first condition is likely, but the second one is not. Indeed, S_{P^-} is likely to be essentially $S_{P_1^-}$, unless the coefficients $c_{n-1}, \ldots, c_q$ are significantly larger than $c_{q-1}, \ldots, c_0$.

Algorithm 16 implements the partial sum trick.

Algorithm 16: `taylorShiftpartialSum`$(p[0\ldots n-1], r[0\ldots s-1], B, \alpha)$

Input: Array $p[0\ldots n-1]$ of size n; an auxiliary array $r[0\ldots s-1]$ of size $s = 2n$; initially $p[0\ldots n-1]$ and $r[0\ldots n-1]$ contain the input coefficients; $p[0\ldots n-1]$ and $r[0\ldots s-1]$ are overwritten during the computations; B is the base size (threshold) and α is a portion of the L1 cache, typically 1/4.

Output: Coefficient array of the polynomial $p(x+1)$ stored in the array $p[0\ldots n-1]$ such that $a[i]$ is the coefficient of the term of degree i.

```
1:  if n ≤ B then
2:      taylorShiftBase(p[0...n-1], r[0...n-1]);
3:  else
4:      m = n/2;
5:      spawn tableauConstruction(p[0...m-1], r[0...m-1], B);
6:      spawn taylorShiftpartialSum(p[m...n-1], r[n...s-1], B, α);
7:      sync;
8:      for i = n to s-1 do
9:          r[i] ← 0;
10:     spawn taylorShiftGeneralDynamic(p[0...m-1], r[m...n-1], B, α);
11:     spawn taylorShiftpartialSum(r[0...m-1], r[n...s-1], B, α);
12:     sync;
13:     addVector(p[m...n-1], r[0...m-1]) ;        /*This is just vector
        addition*/
```

Algorithm 17 combines the techniques of Algorithms 15 and 16 into a single method that we call *Combo*. The idea is to modify the scheme of Algorithm 15 as follows. When the input coefficients $p[m]\ldots p[m+9]$ are sufficiently large compared to the coefficients $p[0]\ldots p[9]$, the partial sum trick is applied, otherwise we follow the original scheme of Algorithm 15. This criterion is based on the estimates of Formula (6.3).

Algorithm 17: `taylorShiftCombo`$(p[0 \dots n-1], q[0 \dots n-1], \alpha)$

Input: $p[0 \dots n-1]$ is the coefficient array (dense representation) of a univariate polynomial $p(x)$ of degree d where $n = d + 1$ ($a[i]$ is the coefficient of the term of degree i); $q[0 \dots n-1]$ is another array of length n, where all coefficients are initially zero; both $p[0 \dots n-1]$ and $q[0 \dots n-1]$ are overwritten; α is a portion of the L1 cache, typically equal to $1/4$.

Output: Coefficient array of the polynomial $p(x+1)$ stored in the array $p[0 \dots n-1]$ such that $a[i]$ is the coefficient of the term of degree i.

1: Randomly choose 10 coefficients from p;
2: Randomly choose 10 coefficients from q;
3: Find the largest coefficient among them;
4: H = Bit size of largest coefficient;
5: $B = \frac{1}{2} - \frac{H}{3} + \frac{(-3+4H^2+3\alpha Z)^{\frac{1}{2}}}{6}$;
6: $m = (\frac{n+1}{2})$;
7: A_a = Average size of 10 coefficients $p[0] \dots p[9]$ in bits;
8: A_b = Average size of 10 coefficients $p[m] \dots p[m+9]$ in bits;
9: **if** $B \leq B_{\min}$ **then**
10: $\quad B = B_{\min}$
11: ;
12: **if** $n \leq B$ **then**
13: $\quad$ `taylorShiftBase`$(p[0 \dots n-1], q[0 \dots n-1])$;
14: **else**
15: $\quad$ **if** $A_a + 2 * m < A_b$ **then**
16: $\quad\quad$ `tableauConstruction`$(p[0 \dots m-1], q[0 \dots m-1], B)$;
17: $\quad\quad$ **spawn** `taylorShiftCombo`$(p[0 \dots m-1], q[m \dots n-1], \alpha)$;
18: $\quad\quad$ **spawn** `taylorShiftCombo`$(p[m \dots n-1], q[0 \dots m-1], \alpha)$;
19: $\quad\quad$ **sync**;
20: $\quad$ **else if** $A_a + 2 * m \approx A_b$ **then**
21: $\quad$ Introduce array r of size $s = 2n$;
22: $\quad$ $r[0 \dots n-1] = q[0 \dots n-1]$;
23: $\quad$ $r[n \dots s-1] = 0$;
24: $\quad$ `taylorShiftpartialSum`$(p[0 \dots n-1], r[0 \dots s-1], B, \alpha)$;

6.3 Experimental results

We have conducted the experiments on the implementation of our various parallelization of Taylor shift computations and VCA Algorithm on the cluster `stegosaurus.csd.uwo.ca`. This machine has one head and 13 compute nodes, called *node-0-0* to *node-0-12*. We have used *node-0-0* whose configuration is: AMD Opteron(tm) Processor 6168, L1 cache size 64KB, L2 cache size 512 KB, number of processors: 48, CPU family: 16, Model: 9, CPU MHz: 800.

We have run both static and dynamic divide and conquer schemes on different polynomial families such as BND [16], CND [16], PSnd, Chebyshev, Mignotte etc. for Taylor shift computation. *BND* and *CND* are well known experimental polynomial families introduced by Jeremy R. Johnson Werner Krandick and Anatole D. Ruslanov in their paper [16]. Let d be the degree of a polynomial in one of those families. The definition of *BND* can be given as:

$$Bnd(x) = dx^n + dx^{n-1} + \cdots + dx + d.$$

The definition of *CND* is:

$$Cnd(x) = x^n + d.$$

The *Mignotte* polynomial is (assuming $n >= 2$):

$$x^n - 2(5x - 1)^2.$$

We define a *Chebyshev* polynomial as follows. If, $T_0 = 1, T_1 = x$ then $T_n = 2T_{n-1} - T_{n-2}$.

We have introduced three other example polynomial families. The definition of *PSnd* can be given as:

$$PSnd(x) = 2 + 2^2 + \cdots + 2^{2(n-1)} + 2^{2n}.$$

Another example polynomial family *PolynomialEx1* is defined as:

$$Pex1(x) = 1 + \cdots + dx^{n/2} + \cdots + dx^n.$$

The last example polynomial family that we have introduced is *PolynomialEx2*, defined as:

$$Pex2(x) = 1 + \cdots + dx^k + \cdots + x^n.$$

where, $2n/3 \leq k < n$.

6.3.1 Divide and conquer: static vs dynamic

We have run both the static and dynamic divide and conquer schemes for the different polynomial families defined above. We measured their timings for both 1 processor and 48 processors, then calculated speed up. The results are shown in the Table 6.1. By static and dynamic approaches we mean Algorithms 6 and 17, respectively.

From Table 6.1, we see that, for different degree polynomials in different polynomial families, the dynamic approach is faster (in terms of running time) than the static approach. Moreover, it has better speed up than the static method.

To show the comparative results in a better way, we provide a plot, see Figure 6.3 which shows a comparison *speedup* and *parallelism* for the static and dynamic approaches, for the input CND polynomial of degree 25000, running on our AMD machine.

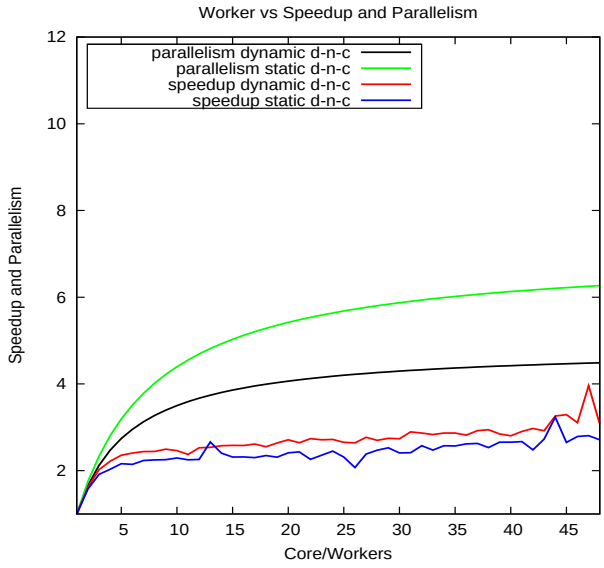


Figure 6.3: Speedup and parallelism comparison of static and dynamic divide and conquer for degree 25000 `Cnd` polynomial.

From Figure 6.3 it is clearly seen that the dynamic approach has better speed up comparing to the static method. Although the parallelism (which is a theoretical measure) seems higher for static divide and conquer, more cache hits bring comparatively better performance to the dynamic method.

From Figure 6.4 it is clearly seen that for different number of processors the *wall*

Polynomial family	Method	n	k	Processor(s)		Speedup
				48-Processors	1-Processor	
Bnd [16]	Static DnC	5000	5000	1.193	2.184	1.83
	Dynamic DnC	5000	5000	1.160	2.187	1.88
	Static DnC	10000	10000	5.391	16.774	3.11
	Dynamic DnC	10000	10000	5.433	16.799	3.09
	Static DnC	25000	25000	44.161	250.16	5.66
	Dynamic DnC	25000	25000	43.188	250.942	5.81
Cnd [16]	Static DnC	5000	5000	0.966	0.939	0.97
	Dynamic DnC	5000	5000	0.893	0.984	1.10
	Static DnC	10000	10000	4.39	6.285	1.43
	Dynamic DnC	10000	10000	3.978	6.646	1.67
	Static DnC	25000	25000	32.523	88.158	2.71
	Dynamic DnC	25000	25000	29.356	90.631	3.08
PSnd	Static DnC	5000	NA	0.976	0.94	0.96
	Dynamic DnC	5000	NA	0.903	0.994	1.10
	Static DnC	10000	NA	4.435	6.334	1.42
	Dynamic DnC	10000	NA	4.008	6.672	1.66
	Static DnC	25000	NA	31.576	88.363	2.79
	Dynamic DnC	25000	NA	30.352	91.015	2.99
Chebyshev	Static DnC	5000	NA	0.951	2.063	2.16
	Dynamic DnC	5000	NA	0.982	2.082	2.12
	Static DnC	10000	NA	4.737	15.306	3.23
	Dynamic DnC	10000	NA	4.734	15.224	3.21
Mignotte	Static DnC	5000	NA	1.000	0.93	.93
	Dynamic DnC	5000	NA	0.906	0.988	1.09
	Static DnC	10000	NA	4.488	6.29	1.40
	Dynamic DnC	10000	NA	3.974	6.628	1.66
PolynomialEx1	Static DnC	5000	5000	1.037	1.166	1.12
	Dynamic DnC	5000	5000	1.000	1.163	1.16
	Static DnC	10000	10000	4.467	8.202	1.83
	Dynamic DnC	10000	10000	4.436	8.16	1.83
	Static DnC	25000	25000	32.629	116.977	3.58
	Dynamic DnC	25000	25000	31.874	117.56	3.68
PolynomialEx2	Static DnC	5000	5000	1.004	0.932	0.92
	Dynamic DnC	5000	5000	0.869	0.984	1.13
	Static DnC	10000	10000	4.586	6.292	1.37
	Dynamic DnC	10000	10000	4.106	6.642	1.61
	Static DnC	25000	25000	33.097	88.183	2.66
	Dynamic DnC	25000	25000	30.478	90.771	2.97

Table 6.1: Taylor shift computation (timings in seconds) [Platform:stegosaurus cluster node-0-0, Static base size=50].

time of the dynamic divide and conquer approach is better than for one processor. To generate this graph, we took *trial results* generated by *Cilkview* for 1 to 48 Processor(s) for input CND polynomial (degree 25000) running on our AMD machine.

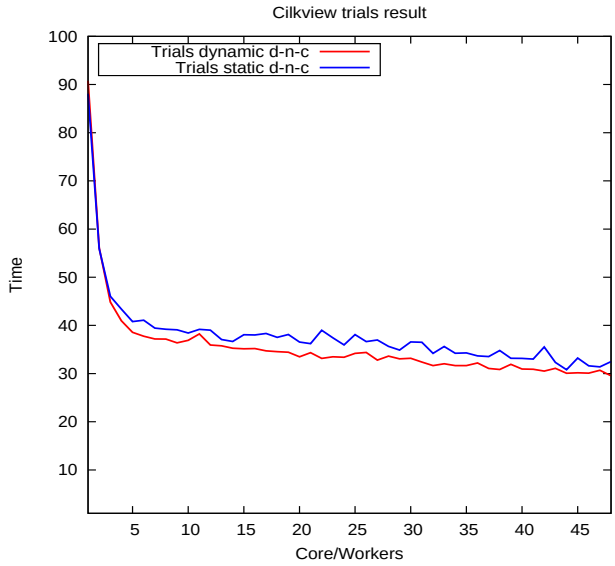


Figure 6.4: Real timing comparison between static and dynamic divide and conquer for degree 25000 `Cnd` polynomial.

6.3.2 Comparative results: Partial sum vs divide and conquer

We have stated in Section 6.2 that there are examples where we have significantly large polynomial coefficients at the end of the Pascal Triangle calculation. In those cases we will gain better performance if we follow the partial sum trick. To justify this, we have made a special polynomial family, (we named it *PSnd*) for which we have made a comparative study between the static divide and conquer approach, the dynamic divide and conquer approach and the divide and conquer approach using the partial sum trick.

The comparative results for partial sum, static and dynamic divide and conquer for *PSnd* polynomials of different degree are shown in Table 6.2. We have run the experiment on 1 processor and 48 processors of our AMD node.

Figure 6.5 shows the comparison graph for speedup between static divide and conquer, dynamic divide and conquer and partial sum for *PSnd* polynomial of degree 25000. From the graph it is clearly seen that partial sum trick has better speedup for this polynomial family.

Polynomial family	Method	n	k	Processor(s)		Speedup
				48-Processors	1-Processor	
PSnd	Static DnC	5000	5000	0.976	0.94	0.96
	Dynamic DnC	5000	5000	0.903	0.994	1.10
	Partial sum	5000	5000	0.808	1.288	1.59
	Static DnC	10000	10000	4.435	6.334	1.42
	Dynamic DnC	10000	10000	4.008	6.672	1.66
	Partial Sum	10000	10000	3.731	8.487	2.27
	Static DnC	25000	25000	31.576	88.363	2.79
	Dynamic DnC	25000	25000	30.352	91.015	2.99
	Partial sum	25000	25000	28.441	113.484	3.99

Table 6.2: Taylor shift computation (timings in seconds) [Platform:stegosaurus cluster node-0-0, Static base size=50].

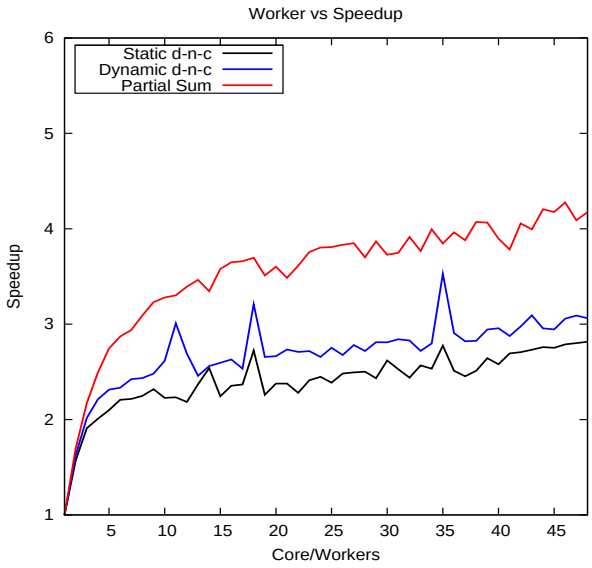


Figure 6.5: Speedup comparison between static divide and conquer, dynamic divide and conquer and Partial sum for degree 25000 `PSnd` polynomial.

Polynomial family	Method	n	Processor(s)			Speedup	
			48-Procs	12-Procs	1-Proc	12-Procs	48-Procs
Chebyshev	Static DnC	400	23.411	24.846	134.821	5.42	5.76
	Dynamic DnC	400	23.584	24.977	135.035	5.40	5.73
	Static DnC	500	45.524	62.788	413.009	6.58	9.07
	Dynamic DnC	500	43.722	62.526	413.439	6.61	9.46
Mignotte	Static DnC	400	73.092	79.138	206.636	2.61	2.82
	Dynamic DnC	400	73.673	79.242	206.738	2.60	2.81

Table 6.3: Real root isolation (timing in seconds) [Platform:stegosaurus cluster node-0-0, Static base size=50].

Polynomial family	Method	n	Processor(s)			Speedup	
			48-Procs	12-Procs	1-Proc	12-Procs	48-Procs
Hilbert 16	Static DnC	426	163.094	200.484	1119.02	5.58	6.86
	Dynamic DnC	426	162.13	200.068	1120.19	5.60	6.91

Table 6.4: Real root isolation Hilbert-16 (timing in seconds) [Platform:stegosaurus cluster node-0-0, Static base size=50].

6.3.3 Root isolation results

We have tried root isolation for the well known polynomials *Chebyshev* and *Mignotte* on the same cluster node. The real root isolation results for *Chebyshev* and *Mignotte* polynomials (of different degrees) for the *static divide and conquer* and *Dynamic divide and conquer* are shown in Table 6.3.

From the above table we can see that the dynamic divide and conquer brings better speedup for isolating real roots of the Chebyshev polynomial when degree is high (say, 500). One can observe from the same table that we have reached our maximum speedup when we utilize 12 processors.

6.3.4 Root isolation results: Hilbert-16 polynomial family

In addition, we have used our software to isolate the real roots of a well known large polynomial called `Hilbert-16`. This polynomial has degree 426, and the bit size of its coefficients is 1900. It is available at `http://www.orcca.on.ca/~cchen/ammcs2011.txt`. Experimental results are shown in Table 6.4.

From Table 6.4 we can see that, for the *Hilbert 16* polynomial, the dynamic divide and conquer approach performs better than the static one.

Chapter 7

Blocking Taylor shift: Dynamic approach

This chapter describes implementation techniques for Taylor shift computations using the blocking scheme. In Chapter 4, our algorithms were using a block of fixed order B throughout the entire computation of the Pascal Triangle. Moreover, this order was the same for all test examples in the experiment of [6], namely 50.

Based on the results of Chapter 5, in particular Section 5.4, we believe that

1. by virtue of Formula (5.19), the *initial* order B should be calculated from the maximum size H of the coefficients of the input polynomial and the L1 cache size Z, and

2. the order B of the blocks can be reduced, say divided by 2, after s/B bands, where s is given by Formula (5.21).

However, from Proposition 5, one should be aware that dividing the block order increases the burdened span faster than it increases the non-burdened span. Roughly speaking, this means that, parallelization overheads may grow faster than the parallelization benefits.

Sections 7.1 and 7.2 describe algorithms which implement the above two techniques. Experimental results based on these techniques are reported in section 7.3.

7.1 Dynamic granularity in blocking strategy

Repeating from Chapter 4, where we have discussed blocking methods for Taylor shift, where we did *not* consider the growth of the coefficients and had fixed base size

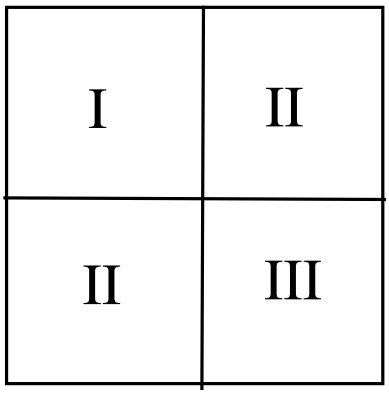


Figure 7.1: Simple static blocking strategy.

for the entire process. This method has several limitations, for instance, this method is *not* cache friendly and there is room for gaining more parallelism.

We would like to consider the following issues:

1. the growth rate of coefficients,
2. making the blocks more cache friendly (so that it has less cache misses), and
3. gaining more parallelism.

Our plan is to modify static block method so that the blocks become cache aware.

For convenience, we rephrase our definitions that we gave in Chapter 6. Let $p(x) \in \mathbb{Q}[x]$ be a polynomial of degree d (where $n = d + 1$) whose coefficients are stored in an array $a[0 \dots n-1]$, where $c[i]$ is the coefficient of the term of degree i. We denote H as coefficient size in bits. Let us denote the cache as Z and α be a portion of the L1 cache, typically equal to $1/4$, for the reasons noted in Section 5.4.

From the explanation noted in Section 5.1, we observed that at the beginning of computation, if H is small then we can take many coefficients of size H inside a block without any cache misses. That makes our B bigger, the explanation can be found in Section 5.4. During the computation H grows and size of B needs be reduced to avoid cache misses. There is a more detailed discussion on Section 7.2. While our block size is getting smaller, we are gaining more parallelism.

In this picture, we show that a region can be divided into 4 regions, where the region I is of size 2^0 and region II and III is of size 2^2. Here, by 'size', we mean work in that region. Assume that each of the regions have n coefficients. Region I has small coefficients. Region II and region III have bigger coefficients which make them 4 times larger than region I.

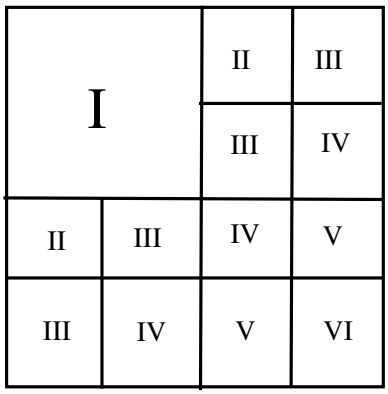


Figure 7.2: New blocking strategy.

According to the static blocking strategy, only region II works in parallel. But we can see here, we can divide region II and region III into 4 more blocks.

We can see from Figure 7.1 that 4 blocks were completed in 3 steps and we can use at most 2 workers in parallel for region II. But, we have the opportunity to introduce more workers if we follow the strategy shown in Figure 7.2.

Figure 7.2 shows that region II and region III are divided into 4 more blocks. The whole problem now has 13 blocks and they are computed in 6 parallel steps, where step II can be computed with 2 workers, step III with 4 workers, step IV with 3 workers and step V with 2 workers. So, this technique gives us opportunity to introduce more workers and achieve more parallelism.

7.2 Cache friendly dynamic blocking

In this section we discuss implementation techniques for cache friendly dynamic blocking.

- At first we compute B based on initial H and the $L1$ cache size of underlying machine architecture we are working on. The function to compute B is already discussed in Chapter 5 as Formula (5.19).

- If B is bigger than n, then the whole problem fits inside the cache and it can be treated as just one triangle. There is no need to divide it into blocks. The entire problem is computed serially inside the cache.

- If B is smaller than n, then we will find the number of diagonal rows we can proceed with this B before B needs to be reduced. Because $B \times B$ ensures block

of inputs can be loaded without any cache miss except cold miss and we want to know the number of rows s, after that we can no longer compute a square block without incurring cache misses other than cold misses. For a given B, H, and $L1$ the process to get s is given by formula 5.21. We continue computation with block order B until s/B bands. From now on we will call these bands *Steps*.

- Just like static blocking, there are two cases in dynamic blocking, *Regular case* and *Irregular Case*. Regular case means there will be only tableau and triangle blocks whereas the irregular case, there will be special block like trapezoid. This will only happen when n is not divisible by B.

- If we multiply B by the number of steps, that gives us the number of inputs we can proceed with block B and at which point we need to change B. If *steps* multiplied by B is bigger than n, then we can say that entire calculation does not need to reduce B and each block will fit inside cache.

- We always know in those steps that, there is a final step which will be triangle (Taylor), one possible step for trapezoid (Polygon) (if $n \nmid B$) and other steps should be square (Tableau).

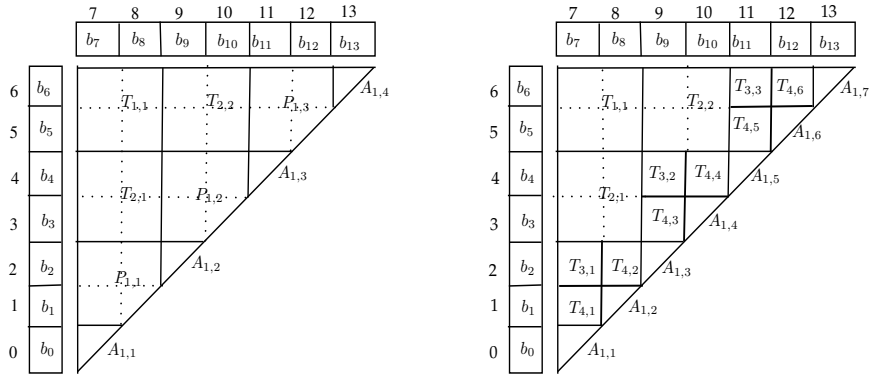
- There is a serious implementation challenge for dynamic blocks, namely, we can not tell at beginning of dynamic block if it will have polygon blocks or not. Indeed, in the static block we could detect the presence of polygons at the very beginning. The reason is, the presence of a polygon in the program strongly depends on change of B. As an example, for input size $n = 7$ and Base size $B = 2$, we initially guess for static block that there are two steps of tableau of base 2, one step of polygon of base 2 and 1 step of triangle of base 1. But for dynamic blocking strategy, if the base size becomes 1 after 2 steps of tableau, then the polygon part vanishes and instead of the polygon there will be 2 steps of tableau of base 1 and one final step of triangle. Figure 7.3 shows how the polygon parts $P_{1,1}, P_{1,1}$ and $P_{1,3}$ of sub-figure (a) change to 2 steps of tableau parts $T_{3,1}, T_{3,2}, T_{3,3}$ and $T_{4,1}, T_{4,2}, T_{4,3}, T_{4,4}, T_{4,5}$, and $T_{4,6}$ in sub-figure(b). We resolve this issue by checking for polygon's presence at the point when we change B.

- We proceed, doing tableau from beginning until the point the predicted number of steps says we should reduce our B. When we reach that point, we check if

we really need to reduce B or not. If current H still fits inside cache, then we are fine and we do not need to bother reducing B. But if H does not fit inside cache then we reduce our B to $B/2$.

- With this reduced B we again find out the number of steps.

  We will continue to do the same process until the entire computation is done.



(a) When B is unchanged **(b) When B becomes $B/2$**

Figure 7.3: Case illustration for dynamic blocking strategy.

The following algorithms show the way to implement a dynamic block. Algorithm 18 and Algorithm 19 are the main controlling part. Algorithm 21 works if B remains unchanged. Whether B should be reduced or not is decided by Algorithm 22. Algorithm 23 returns the number of steps that can be continued before a change on B. When B is reduced Algorithm 20 controls the computation. After a tableau is completed, Algorithm 24 controls final steps such as polygons and triangles. A calculation procedure for polygon is controlled by Algorithm 25 and triangles are controlled by Algorithm 26.

Algorithm 18 applied to (a, n, α) computes and stores the coefficients of $p(x+1)$ in the array $a[0 \ldots n-1]$. Algorithm 18 is based on the blocking scheme discussed in Chapter 4. However, and on the contrary to the static block algorithm, the base case (or threshold) B is determined dynamically. Algorithm 18 introduces a new array $b[0 \ldots m-1]$ of size $m = 2n$. In the context of Algorithm 18, we use Formula (5.19) and like we have discussed in Section 6.1, we do not compute the maximum absolute value H of an initial coefficient, that is, the maximum absolute value of a coefficient in b. Rather we use a statistical approach: pick randomly some coefficients (say 100) in $b[0 \ldots n-1]$, then use them to estimate H.

We know from Formula (5.14) that, when B is too small, the burdened span can be significantly larger than the span, thus reducing performances. In Algorithm 18, we refer to this minimum value of B as B_{min}.

Before calling the dynamic block procedure, which is, basically, implemented in Algorithm 19, Algorithm 18 uses context from Section 5.4 to find the number of steps after that a square $B \times B$ block will have cache misses.

Algorithm 18: `dynamicBlock`$(a[0 \dots n-1], n, \alpha)$

Input: $a[0 \dots n-1]$ is the coefficient array (dense representation) of a univariate polynomial $p(x)$ of degree d where $n = d + 1$ ($c[i]$ is the coefficient of the term of degree i); $a[0 \dots n-1]$ is overwritten during computation and α is a portion of the L1 cache, typically equal to 1/4.

Output: Coefficient array of the polynomial $p(x+1)$ stored in the array $a[0 \dots n-1]$, such that $c[i]$ is coefficient of term of degree i.

1: $m = 2n$;
2: $b[0, \dots, n-1] = a[0, \dots, n-1]$;
3: $b[n, \dots, m-1] = 0$;
4: Pickup some coefficients from b and get average size of them;
5: $H =$ Average coefficient size in bit;
6: $B = \frac{1}{2} - \frac{H}{3} + \frac{(-3+4H^2+3\alpha Z)^{\frac{1}{2}}}{6}$;
7: **if** $B < B_{\min}$ **then**
8: $\quad B = B_{\min}$;
9: **if** $n <= B$ **then**
10: $\quad B = n$;
11: $\quad$ `taylorBaseBlock`$(b[0 \dots n-1], B)$;
12: $\quad a[0, \dots, n-1] = b[0, \dots, n-1]$; `/*copy results from b to a*/`
13: **else**
14: $\quad S =$ `findSteps`(B, H, α);
15: $\quad$ `dynamicSubBlock`$(b[0 \dots m-1], n, B, S)$;
16: **return** $a[0 \dots n-1]$;

Algorithm 19 makes all required decisions to implement a dynamic blocking scheme. With the number of steps provided to it by Algorithm 18, it proceeds computing blocks by calling Algorithm 21. Whenever it reaches the last step, it makes a decision about reducing B by calling Algorithm 22. If B is reduced, computation gets a little bit tricky and Algorithm 20 is called to do this job.

Algorithm 19: `dynamicSubBlock`$(b[0 \dots m-1], n, B, S)$

Input: $b[0 \dots m-1]$ is the array containing coefficients of an univariate polynomial $p(x)$ of degree d ($n = d+1$), where $b[i]$ ($0 \leq i \leq n-1$), is the coefficient of the term of degree i ; B is the base size (threshold) and S is the number of steps.

Output: Take decision on calling different sub-functions and calls Algorithm 20, 21, 22 and 24.

1: $\overline{B} = B$;
2: $w = 0$;
3: $e = 0$;
`/* e is the number of blocks at each step */`
4: $\overline{n} = n$;
5: **while** ***TRUE*** **do**
6: $j = 1$;
7: **if** $\overline{n} >= SB$ **then**
8: $\overline{S} = S$;
9: **else**
10: $\overline{S} = \frac{\overline{n}}{B}$;
11: **if** $B = \frac{\overline{B}}{2}$ **then**
12: `newBaseHalved`$(b[0 \dots m-1], n, B, \overline{S}, w, e)$;
`/* this condition works when base size is reduced */`
13: $\overline{B} = B$;
14: **else**
15: `baseUnchanged`$(b[0 \dots m-1], n, B, \overline{S}, w, e)$;
`/* this condition works when base size is unchanged */`
16: $w = w + \overline{S}$;
`/* w says how many steps are done. */`
17: $\overline{n} = \overline{n} - (\overline{S}B)$;
`/* number of inputs that are left to do */`
18: **if** $\overline{S} \neq S$ **then**
19: break;
20: **else**
21: `baseDecrementDecision`$(b[0 \dots m-1], w, B, \alpha)$;
`/* this function returns new base size and new` S `*/`
22: `finalCases`$(b[0 \dots m-1], n, w, \overline{n}, B, e, \overline{S}, \overline{B})$;

Algorithm 20: `newBaseHalved`$(b[0 \ldots m-1], n, B, \overline{S}, w, e)$

Input: $b[0 \ldots m-1]$ is an array of size m; $b[0 \ldots m-1]$ is overwritten during computation; w is the number of computed steps; B is the base size (threshold); number of steps $\overline{S}$ and e is the number of blocks at current step.

Output: Updated array $b[0 \ldots m-1]$ and e (the number of blocks at current step).

```
1:  for j = 1 to S̄ do
2:      if j = 1 then
3:          ī = 4;
4:      else
5:          ī = 2;
6:      if j = 2 then
7:          e = 2e;
8:      else
9:          e = e + 1;
10:     for i = 0 to e − 1 do
11:         t = n − w − (jB) + (īBi);
12:         spawn tableauBaseBlock(b[t . . . t + B], B);
13:     sync;
14: return b[0 . . . m − 1], e ;
```

Algorithm 21: `baseUnchanged`$(b[0 \ldots m-1], n, B, \overline{S}, w, e)$

Input: $b[0 \ldots m-1]$ is an array of size m; $b[0 \ldots m-1]$ is overwritten during computation; w is the number of computed steps; B is the base size (threshold); number of steps $\overline{S}$ and e is the number of blocks at current step.

Output: Updated array $b[0 \ldots m-1]$ and e (the number of blocks at current step).

```
1: for j = 1 to S̄ do
2:     e = e + 1;
3:     for i = 0 to e − 1 do
4:         t = n − w − (jB) + (2iB);
5:         spawn tableauBaseBlock(b[t ... t + B], B]);
6:     sync;
7: return b[0 ... m − 1], e;
```

For tableau construction of a block we follow the Algorithm 11, `tableauBaseBlock`, discussed in Chapter 4.

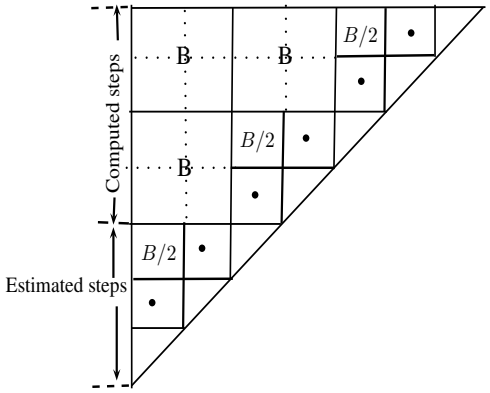


Figure 7.4: Dynamic blocking strategy.

Algorithm 22: `baseDecrementDecision`$(b[0 \dots m-1], w, B, \alpha)$

Input: $b[0 \dots m-1]$ is an array of size m; w is the number of steps until which computation is done; B is the block order (threshold); α is a portion of the L1 cache, typically equal to 1/4.

Output: Updated B (block order) and S (number of steps).

Choose the 100 most recently computed coefficients and 100 input coefficients and find the largest one;
H = largest coefficient in bitsize ;
$s' =$ `findSteps`(B, H, α);
if $s' \leq 0$ **then**
    `/* it means current block size can not contain` H `size coefficients without cache misses. */`
    $B = \frac{B}{2}$;
    $S =$ `findSteps`(B, H, α);
else
    $B = B$;
    `/* with current block size current coefficients fits inside cache */`
    $S = s'$;
return B, S;

Algorithm 23: `findSteps`(B, H, α)

Input: B is the block order, H is coefficient size in bits, α is a portion of the L1 cache, typically equal to 1/4.

Output: Number of steps S until which a $B \times B$ block will fit inside cache.

$S = 0$;
$h = 2HB - H + 3B^2 - 3B + 1$;
while $h < \alpha Z$ **do**
    $S = S + 1$;
    $h = 2HB - H + 2SB^2 - SB + 3B^2 - 3B + 1$;
return S;

Algorithm 24: `finalCases`$(b[0 \dots m-1], n, w, \overline{n}, B, e, \overline{S}, \overline{B})$

Input: $b[0 \dots m-1]$ is an array of size m; $b[0 \dots m-1]$ is overwritten during computation; w is the number of steps until which computation is done; $\overline{n}$ is the number of inputs that are left to compute; B is the block order (threshold); e is the number of blocks at current step; $\overline{S}$ is the number of steps proceeded with B; and $\overline{B}$ is size of base before change.

Output: Calls Algorithm 25 and Algorithm 26 to do last steps of Pascal Triangle.

1: $r = \overline{n}$ **mod** B;
2: **if** $\overline{S} = 1$ ***AND*** $B = \frac{\overline{B}}{2}$ **then**
3: $\quad e = 2e$;
4: **else**
5: $\quad e = e + 1$;
6: **if** $r > 0$ **then**
`/* when there is reminder, special case like polygons happen */`
7: `polygonCase`$(b[0 \dots m-1], n, w, B, r, e)$;
8: **else**
9: `triangleCase`$(b[0 \dots m-1], B, e)$;
`/* when r=0 */`

When all the square regions are computed and only the last steps like polygons or triangles are left to do, then Algorithm 24 is called. This algorithm makes a decision on whether to do polygons or triangles for last steps and calls Algorithm 25 and Algorithm 26 accordingly. Algorithm 25 computes last steps if there are polygons present. Algorithm 26 works if there are only triangles at the last step.

Algorithm 25: `polygonCase`$(b[0 \dots m-1], n, w, B, r, e)$

Input: $b[0 \dots m-1]$ is an array of size m; $b[0 \dots m-1]$ is overwritten during computation; w is the number of steps until which computation is done; B is the block order (threshold); e is the number of blocks for polygon step; r is the size of triangles at last step and e is number of blocks at polygon step.

Output: Coefficient array of the polynomial $p(x+1)$ stored in the array $a[0 \dots n-1]$ such that $c[i]$ is the coefficient of the term of degree i.

1: $k = B - r - 1$;
2: **if** $k > 0$ **then**
3: $\quad k = k$;
4: **else**
5: $\quad k = 1$;
`/* polygon */`
6: **for** $i = 0$ to $e - 1$ **do**
7: $\quad t = n - w + (2iB)$;
8: $\quad$ **spawn** `polygonBase`$(b[t \dots t + B], B, r, k)$;
9: **sync**;
`/* triangle */`
10: **for** $i = 0$ to e **do**
11: $\quad$ **spawn** `taylorBaseBlock`$(b[2iB \dots 2iB + r], r)$;
12: **sync**;
`/* copy triangle */`
13: **for** $i = 0$ to e **do**
14: $\quad$ **for** $j = 0$ to $r - 1$ **do**
15: $\quad\quad a[j + iB] = b[j + 2iB]$;
`/* copy polygons */`
16: **for** $i = 0$ to e **do**
17: $\quad$ **for** $j = 0$ to $B - r - 1$ **do**
18: $\quad\quad a[iB - j] = b[2iB - j]$;
19: $\quad a[(i-1)B + r] = b[2((i-1)B + r)]$;
20: **return** $a[0 \dots n-1]$;

Algorithm 26: `triangleCase`$(b[0 \dots m-1], B, e)$

Input: $b[0 \dots m-1]$ is an array of size m; $b[0 \dots m-1]$ is overwritten during computation and e is the number of blocks for triangle step.

Output: Coefficient array of the polynomial $p(x+1)$ stored in the array $a[0 \dots n-1]$ such that $c[i]$ is the coefficient of the term of degree i.

```
1: for i = 0 to e − 1 do
2:     spawn taylorBaseBlock(b[2iB ... 2iB + B], B);
3: sync;
   /* copy from b to a                                          */
4: for i = 0 to e − 1 do
5:     for j = 0 to B − 1 do
6:         a[j + iB] = b[j + 2iB];
7: return a[0 ... n − 1] ;
```

For computing the triangle case, we will follow Algorithm 12.

7.3 Experimental results

We conducted the experiments for the implementation of our blocking method of Taylor shift computations and VCA Algorithms on the cluster `stegosaurus.csd.uwo.ca`. We already have described about this machine and configuration of one of its compute node-0-0 in Section 6.3. For our experimentation we have also used another node *node-0-2* whose configuration is: Intel(R) Xeon(R) CPU X5650 with 2.67GHz processor, L1 cache size 32KB, L2 cache size 12288 KB, Number of Processors: 24, CPU family: 6, Model: 44, CPU MHz: 1600.

7.3.1 Block: static vs dynamic

We have run both static and dynamic block schemes on different polynomial families such as BND [16], CND [16], PSnd, Chebyshev, Mignotte etc. for the Taylor shift computation. Definitions of these polynomial families are already specified in Section 6.3. We measure their timings for both 1 processor and 48 processors and calculate speed up for both *AMD* and *Intel* machines. By static and dynamic approaches we refer to Algorithms 10 and 18, respectively. Experimental results running on the Intel machine are shown in Table 7.1.

Experimental results running on the AMD machine are shown in Table 7.2.

Polynomial family	Method	n	k	Processor(s)		Speedup
				12-Procs	1-Proc	
Bnd [16]	Static block	5000	5000	0.506	1.505	2.97
	Dynamic block	5000	5000	0.493	1.505	3.05
	Static block	10000	10000	2.406	11.372	4.72
	Dynamic block	10000	10000	2.223	11.378	5.11
	Static block	25000	25000	23.551	162.654	6.90
	Dynamic block	25000	25000	21.102	162.546	7.70
Cnd [16]	Static block	5000	5000	0.401	0.628	1.56
	Dynamic block	5000	5000	0.401	0.63	1.57
	Static block	10000	10000	1.832	4.361	2.38
	Dynamic block	10000	10000	1.705	4.262	2.49
	Static block	25000	25000	14.277	60.769	4.25
	Dynamic block	25000	25000	13.493	59.743	4.42
PSnd	Static block	5000	NA	0.427	0.637	1.49
	Dynamic block	5000	NA	0.401	0.635	1.58
	Static block	10000	NA	1.798	4.381	2.43
	Dynamic block	10000	NA	1.699	4.288	2.52
	Static block	25000	NA	15.319	60.897	3.97
	Dynamic block	25000	NA	12.971	59.2	4.56
Chebyshev	Static block	5000	NA	0.403	1.392	3.45
	Dynamic block	5000	NA	0.363	1.391	3.83
	Static block	10000	NA	1.964	10.735	5.46
	Dynamic block	10000	NA	1.712	10.742	6.27
Mignotte	Static block	5000	NA	0.406	0.638	1.57
	Dynamic block	5000	NA	0.393	0.632	1.60
	Static block	10000	NA	1.902	4.335	2.27
	Dynamic block	10000	NA	1.765	4.249	2.40
PolynomialEx1	Static block	5000	5000	0.414	0.788	1.90
	Dynamic block	5000	5000	0.406	0.779	1.91
	Static block	10000	10000	1.941	5.699	2.93
	Dynamic block	10000	10000	1.865	5.571	2.98
	Static block	25000	25000	15.65	78.68	5.02
	Dynamic block	25000	25000	14.422	77.371	5.36
PolynomialEx2	Static block	5000	5000	0.384	0.627	1.63
	Dynamic block	5000	5000	0.399	0.63	1.57
	Static block	10000	10000	1.847	4.37	2.36
	Dynamic block	10000	10000	1.748	4.248	2.43
	Static block	25000	25000	14.793	61.074	4.12
	Dynamic block	25000	25000	12.788	59.821	4.67

Table 7.1: Taylor shift computation (timings in seconds) [Platform:stegosaurus cluster node-0-2, Static block order=25].

From both Table 7.1 and Table 7.2, we can see that, for different degree polynomials in different polynomial families, the dynamic blocking method is faster than the static approach (in terms of running time). Moreover, it has better speed up than the static method.

To show the comparative results in a better way, we provide a plot, see Figure 7.5

Polynomial family	Method	n	k	Processor(s)		Speedup
				12-Procs	1-Proc	
Bnd [16]	Static block	5000	5000	1.893	2.447	1.29
	Dynamic block	5000	5000	1.703	2.461	1.44
	Static block	10000	10000	8.269	18.819	2.27
	Dynamic block	10000	10000	8.018	18.877	2.35
	Static block	25000	25000	77.925	287.137	3.68
	Dynamic block	25000	25000	66.038	286.979	4.34
Cnd [16]	Static block	5000	5000	1.392	1.004	0.72
	Dynamic block	5000	5000	1.388	1.112	0.80
	Static block	10000	10000	6.996	6.976	0.99
	Dynamic block	10000	10000	5.641	7.363	1.30
	Static block	25000	25000	49.911	98.978	1.98
	Dynamic block	25000	25000	48.153	99.959	2.07
PSnd	Static block	5000	NA	1.472	1.005	0.68
	Dynamic block	5000	NA	1.261	1.111	0.88
	Static block	10000	NA	6.643	6.994	1.05
	Dynamic block	10000	NA	6.597	7.423	1.12
	Static block	25000	NA	56.534	99.808	1.76
	Dynamic block	25000	NA	48.286	99.909	2.06
Chebyshev	Static block	5000	NA	1.332	2.266	1.70
	Dynamic block	5000	NA	1.459	2.272	1.55
	Static block	10000	NA	6.673	17.168	2.57
	Dynamic block	10000	NA	5.818	17.144	2.94
Mignotte	Static block	5000	NA	1.508	0.984	0.65
	Dynamic block	5000	NA	1.474	1.094	0.74
	Static block	10000	NA	6.367	6.954	1.09
	Dynamic block	10000	NA	6.868	7.355	1.07
PolynomialEx1	Static block	5000	5000	1.552	1.259	0.81
	Dynamic block	5000	5000	1.445	1.325	0.91
	Static block	10000	10000	7.153	9.048	1.26
	Dynamic block	10000	10000	6.028	9.386	1.55
	Static block	25000	25000	57.467	131.709	2.29
	Dynamic block	25000	25000	52.915	132.713	2.50
PolynomialEx2	Static block	5000	5000	1.483	0.988	0.66
	Dynamic block	5000	5000	1.364	1.087	0.79
	Static block	10000	10000	6.343	6.908	1.08
	Dynamic block	10000	10000	6.95	7.371	1.06
	Static block	25000	25000	50.193	98.866	1.96
	Dynamic block	25000	25000	47.27	99.505	2.10

Table 7.2: Taylor shift computation (timings in seconds) [Platform:stegosaurus cluster node-0-0, Static block order=50].

which shows a comparison of *speedup* and *parallelism* for the static and dynamic approaches, for input 25000 degree BND polynomial running on our AMD machine.

Figure 7.6 shows wall timing comparison between 1-processor and 12-processors on AMD machine for different degree BND polynomials.

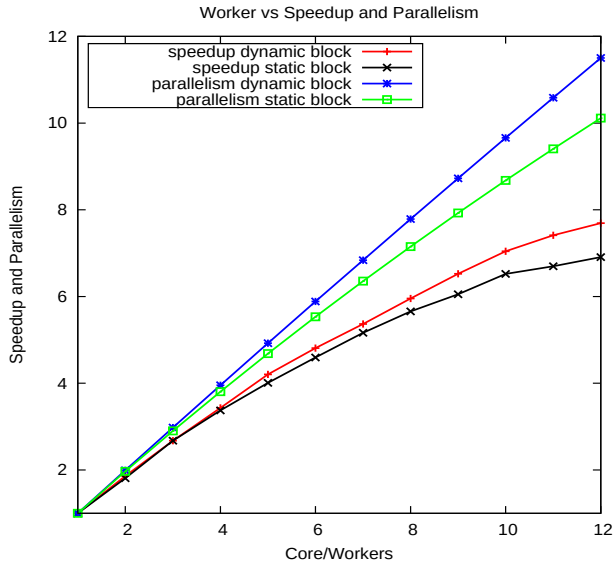


Figure 7.5: Speedup and parallelism comparison between static and dynamic blocks for degree 25000 `Bnd` polynomial.

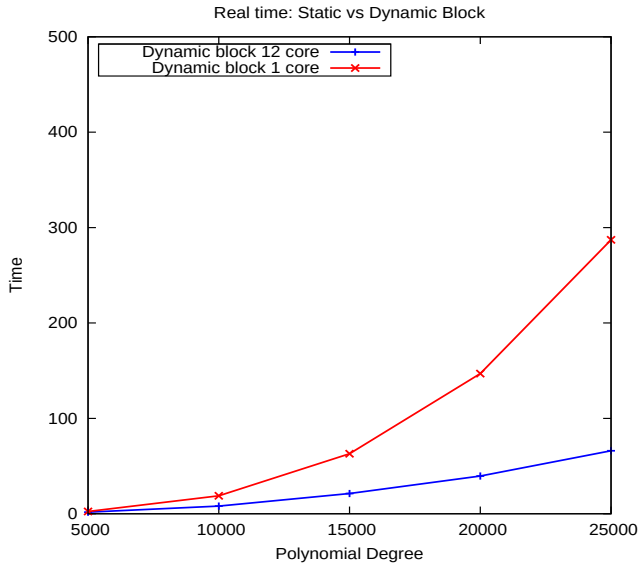


Figure 7.6: Running time for dynamic blocking for different degree `Bnd` polynomials on one processor and 12 processors.

7.3.2 Root isolation results

We have tried root isolation for well known polynomials *Chebyshev* and *Mignotte* on *AMD* machine. The root isolation results for *Chebyshev* and *Mignotte* polynomials (of different degree) for static and dynamic block method are shown in Table 7.3.

Polynomial family	Method	n	Processors		Speedup
			12-Processors	1-Processor	
Chebyshev	Static block	400	36.413	152.304	4.18
	Dynamic block	400	35.361	152.439	4.31
	Static block	500	90.348	472.776	5.23
	Dynamic block	500	88.024	473.903	5.38
Mignotte	Static block	400	183.73	241.376	1.31
	Dynamic block	400	183.104	241.244	1.32

Table 7.3: Root isolation (timings in seconds) [Platform:stegosaurus cluster node-0-0, Static block order=50].

Polynomial family	Method	n	Processors		Speedup
			12-Procs	1-Proc	
Hilbert 16	Static Block	426	465.227	1438.91	3.09
	Dynamic Block	426	455.388	1439.54	3.16

Table 7.4: Root isolation Hilbert-16 (timings in seconds) for Static vs Dynamic Block [Platform:stegosaurus cluster node-0-0, Static block order=50]

From the above table we can see that dynamic divide and conquer brings very good speedup for isolating roots of Chebyshev polynomials.

7.3.3 Root isolation results: Hilbert-16 polynomial family

We also experimented isolating the real roots of a well known large polynomial called Hilbert-16. We are repeating the specification of this polynomial from Section 6.3.4. It has degree 426, and the bit size of its coefficients is 1900. It is available for download at `http://www.orcca.on.ca/~cchen/ammcs2011.txt`. Experimental results are shown in Table 7.4.

From Table 7.4 we can see that, for the Hilbert 16 polynomial, the dynamic block performs better than the static one.

Chapter 8

Conclusion

In this thesis, we investigated implementation techniques for multi-threaded real root isolation of univariate polynomials targeting multi-core architectures.

We have improved previous complexity analyses (work, span, burdened span, cache) by taking into account the growth of the intermediate coefficients. This has lead us to develop poly-algorithms which can adapt the granularity of their parallelism dynamically depending on the local size of the data. Experimentation illustrates the effectiveness of this approach.

Despite of these positive results, we believe that we have reached the limit of what can be done to improve multi-threaded real root isolation on multi-cores. Indeed, for this type of algorithm, as input size grows, parallelization overheads on multi-cores increase faster than parallelization benefits!

Bibliography

[1] L. A. Belady. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal*, 5:78–101, 1966.

[2] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in real algebraic geometry*, volume 10 of *Algorithms and Computations in Mathematics*. Springer-Verlag, 2006.

[3] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twentyfifth Annual ACM Symposium on Theory of Computing*, pages 362–371, San Diego, California, May 1993.

[4] Changbo Chen, James H. Davenport, John P. May, Marc Moreno Maza, Bican Xia, and Rong Xiao. Triangular decomposition of semi-algebraic systems. *CoRR*, abs/1002.4784, 2010.

[5] Changbo Chen, Marc Moreno Maza, and Yuzhen Xie. Cache complexity and multicore implementation for univariate real root isolation. *Journal of Physics: Conference Series*, 341(1):012026, 2012.

[6] Changno Chen, Marc Moreno Maza, and Yuzhen Xie. Cache complexity and multicore implementation for univariate real root isolation. *ACM Commun. Comput. Algebra*, 44:97–98, January 2011.

[7] George E. Collins and Alkiviadis G. Akritas. Polynomial real root isolation using Descarte's rule of signs. In *Proceedings of the third ACM symposium on Symbolic and Algebraic Computation*, SYMSAC '76, pages 272–275, New York, NY, USA, 1976. ACM.

[8] Thomas Decker and Werner Krandick. On the Isoefficiency of the Parallel Descartes Method. In Götz Alefeld, Jiri Rohn, Siegfried M. Rump, and Tet-

suro Yamamoto, editors, *Symbolic Algebraic Methods and Verification Methods*, pages 55–67. Springer, 2001.

[9] Arno Eigenwillig, Vikram Sharma, and Chee K. Yap. Almost tight recursion tree bounds for the Descartes method. In *Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, ISSAC '06, pages 71–78, New York, NY, USA, 2006. ACM.

[10] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–297, New York, USA, October 1999.

[11] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN*, 1998.

[12] Matteo Frigo and Volker Strumpen. The memory behavior of cache oblivious stencil computations. *The Journal of Supercomputing*, 39(2):93–112, 2007.

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

[14] Hong Jia-Wei and H. T. Kung. I/O complexity: The red-blue pebble game. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333, New York, NY, USA, 1981. ACM Press.

[15] Jeremy R. Johnson. *Algorithms for Polynomial Real root Isolation.* PhD thesis, The Ohio State University, 1991.

[16] Jeremy R. Johnson, Werner Krandick, and Anatole D. Ruslanov. Architecture-aware classical Taylor shift by 1. In *ISSAC*, pages 200–207, 2005.

[17] Werner Krandick and Kurt Mehlhorn. New bounds for the Descartes method. *SIGSAM Bull.*, 39:94–94, September 2005.

[18] Fabrice Rouillier and Paul Zimmermann. Efficient isolation of polynomial's real roots. *J. Comput. Appl. Math.*, 162:33–50, January 2004.

[19] Michael Sagraloff. On the complexity of real root isolation. *CoRR*, abs/1011.0344, 2010.

[20] Arnold Schönhage. *The fundamental theorem of algebra in terms of computational complexity-preliminary report*. Universität Tübingen, 1982.

[21] A. J. H. Vincent. Sur la résolution des équations numériques. *Journal de Mathématiques Pures et Appliquées*, 1:341–372, 1836.

[22] Joachim von zur Gathen and Jürgen Gerhard. Fast algorithms for Taylor shifts and certain difference equations. In *Proceedings of the 1997 international symposium on Symbolic and Algebraic Computation*, ISSAC '97, pages 40–47, New York, NY, USA, 1997. ACM.

Curriculum Vitae

Name:	**A.B.M. Zunaid Haque**
Post-Secondary Education and Degrees:	The University of Western Ontario London, Ontario, Canada M.Sc. in Computer Science, April 2012 Islamic University of Technology (IUT), OIC Dhaka, Bangladesh B.Sc. in Computer Science and Information Technology, November 2009
Honours and Awards:	The University of Western Ontario Graduate Research Scholarship (WGRS) 2010-2011
Work Experience:	Research Assistant, Teaching Assistant University of Western Ontario, London, Canada September 2010 - April 2012

Publications:

M.A. Mottalib, Md. Safiur Rahman Mahdi, A.B.M. Zunaid Haque, S.M. Al Mamun and Hawlader Abdullah Al-Mamun, Protein Secondary structure Prediction using Feed-Forward Neural network , ICCIT 09. pp. 598 - 602