Cache Complexity and Multicore Implementation for Univariate Real Root Isolation

Changbo Chen, Marc Moreno Maza and Yuzhen Xie

University of Western Ontario, London, Ontario, Canada

E-mail: `cchen252,moreno,yxie@csd.uwo.ca`

Abstract. We present parallel algorithms with optimal cache complexity for the kernel routine of many real root isolation algorithms, namely the *Taylor shift by 1.* We then report on multicore implementation for isolating the real roots of univariate polynomials with integer coefficients based on a classical algorithm due to Vincent, Collins and Akritas. For processing some well-known benchmark examples with sufficiently large size, our software tool reaches linear speedup on an 8-core machine. In addition, we show that our software is able to fully utilize the many cores and the memory space of a 32-core machine to tackle large problems that are out of reach for a desktop implementation.

1. Introduction

Isolating the real roots of a univariate polynomial is a driving subject in computer algebra. On the one hand, it is closely related to fundamental algorithms such as polynomial multiplication and *Taylor shift* [14]. On the other hand, it is one of the particular areas in scientific computing where computer algebra provides unmatchable tools, for example, counting rigorously the number of real roots of a polynomial. Many researchers have studied this problem under various angles from algebraic algorithms to implementation techniques. We refer the reader to the latest paper [13] for a survey of related works on this topic and the references therein.

Today, multicore processors with hierarchical memories have entered the mainstream of computer architectures. However, harvesting the power of multicores is challenging. Besides, understanding the implications of hierarchical memory has become an essential part of performance software engineering. In this work, we revisit a classical algorithm due to Vincent, Collins and Akritas [4] for isolating the real roots of univariate polynomials. We explore the parallelism of this algorithm and design fine-grained parallel algorithms with different strategies for the kernel routine, i.e., Taylor shift, aiming at attaining efficient implementations for polynomial real root isolation on multicores. This paper is an extension of the two-page abstract for our poster [2] presented in ISSAC 2010. Before this work, several other pioneer works [3, 5, 10, 11] have also been conducted on this topic with different techniques. The paper [3] reports on a first attempt on parallelizing real root isolation problem. Their work shows that the amount of parallelism is limited if only the traversal of the binary tree associated with the isolation problem is parallelized. Later works [5, 10, 11] focus on improving Taylor shift via high performance computing techniques. The paper [5] proposes new scheduling algorithms for the pyramid DAG associated with Taylor shifts and reduces the communication complexity overhead to linear. The paper [10, 11] improves the efficiency of Taylor shift computation and real root isolation based on a technique of register tiling.

This paper is organized as follows. Section 2 introduced the sequential algorithm of Vincent-Collins-Akritas and the Taylor shift operation. In Sections 3, 4 and 5, we present our strategies, *divide-and-conquer* and *blocking*, for a fine-grained parallelization of the Taylor shift procedure and a parallel algorithm for univariate real root isolation. We show that both can be optimal in terms of cache complexity. Their implementation face several challenges (scheduling, in place execution) that we address in Sections 6, 7, 8. In the experimentation, in Section 9, we report timings for processing well-known test examples as well as a real-world challenging problem, respectively on 8-core and 32-core machines. We conclude in Section 10 by discussing extensions of this study.

2. The sequential algorithm

Let p be a univariate squarefree polynomial with integer coefficients. To isolate the real roots of p, the Vincent-Collins-Akritas (VCA, for short) algorithm [4] computes a list of disjoint intervals with rational endpoints such that each real root of p belongs to a single interval and each interval contains only one real root of p. Clearly, the problem of isolating the real roots of p and that counting of them are essentially the same. Thus, one algorithm solving one of these problems can be adapted to solve the other. Since the output of the counting problem is simpler, we present, as Algorithm 1 below, the version of the VCA Algorithm solving this latter problem.

The main subroutine of Algorithm 1 is RootsInZeroOne, that is, Algorithm 2, in which the operation of dominant cost is performed at Lines 3 and 7. This computation, which substitutes x for $x+1$ in p_1, is called the *Taylor shift by 1* (or simply Taylor shift) of the polynomial p_1.

Algorithm 1: RealRoots(p)

```
Input: a univariate squarefree polynomial $p$ of degree $d$
Output: the number of real roots of $p$
1 begin
2     Let $k \geq 0$ be an integer such that the absolute value of all the real roots of $p$ is less than or equal to $2^k$;
3     if $x \mid p$ then $m := 1$ else $m := 0$;
4     $p_1 := p(2^k x)$;
5     $p_2 := p_1(-x)$;
6     $m' :=$ RootsInZeroOne($p_1$);
7     $m := m +$ RootsInZeroOne($p_2$);
8     return $m + m'$;
9 end
```

Algorithm 2: RootsInZeroOne(p)

```
Input: a univariate squarefree polynomial $p$ of degree $d$
Output: the number of real roots of $p$ in $(0, 1)$
1 begin
2     $p_1 := x^d p(1/x)$;
3     $p_2 := p_1(x+1)$; //Taylor shift
4     Let $v$ be the number of sign variations of the coefficients of $p_2$;
5     if $v \leq 1$ then 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 $m := 1$ else $m := 0$;
9     $m' :=$ RootsInZeroOne($p_1$);
10    $m := m +$ RootsInZeroOne($p_2$);
11    return $m + m'$;
12 end
```

Executing Lines 4 and 5 in Algorithm 1 and executing Lines 2 and 6 in Algorithm 2 simply require to "traverse the data" and "update the coefficients", thus have a linear cost with respect to the data size. Apart from its recursive calls, the other computational costs of Algorithm 2 come from the Taylor shift which cost is more than linear (in fact quadratic, as we shall see later). Therefore, same for Algorithm 1, most of the work comes from this latter operation, which justifies the effort devoted to implementing it efficiently.

In order to better understand this operation, consider a polynomial $f(x) = a_d x^d + \cdots + a_1 x + a_0$ of degree d. The Taylor shift of f is the polynomial $f_0(x) = f(x+1)$ computed by the *Pascal's triangle* relation (or Horner's rule) $f_i(x) := f_{i+1}(x) \times (x+1) + a_i$ for i successively equal to $d-1, \ldots, 1, 0$ with $f_d(x) := a_d$, as illustrated for $d = 3$ in Example 1.

Example 1 *Let $f(x) = a_3x^3 + a_2x^2 + a_1x + a_0$. We construct the following Pascal's triangle.*

$$\begin{tikzcd}
& 0 \arrow[d] & 0 \arrow[d] & 0 \arrow[d] & 0 \arrow[d] & \\
a_3 \arrow[r] & + \arrow[r] \arrow[d] & + \arrow[r] \arrow[d] & + \arrow[r] \arrow[d] & + \arrow[r] & c_3 \\
a_2 \arrow[r] & + \arrow[r] \arrow[d] & + \arrow[r] \arrow[d] & + \arrow[r] & c_2 & \\
a_1 \arrow[r] & + \arrow[r] \arrow[d] & + \arrow[r] & c_1 & & \\
a_0 \arrow[r] & + \arrow[r] & c_0 & & &
\end{tikzcd}$$

Performing the addition in the diagonal direction *is exactly the same as computing $f(x+1)$ in Horner's rule, which brings $f(x+1) = c_3x^3 + c_2x^2 + c_1x + c_0$.*

3. Parallelization of the Vincent-Collins-Akritas algorithm

A first opportunity for parallel execution appears in Algorithm 1. Indeed, the two calls to RootsInZeroOne can be performed concurrently. Similarly, in Algorithm 2, the two recursive calls can be performed concurrently. If the amount of work was always balanced among the two calls to RootsInZeroOne in Algorithm 1 as well as among the two recursive calls in Algorithm 2, then those concurrent calls would create significant parallelism. However, this situation will rarely happen in practice. Indeed, observe that one call to Algorithm 2 may quickly terminate at Line 5. In fact, Lines 4 and 5 implement the so-called *Descartes's Rule* (see [12] for detail) which is a criterion for detecting that a polynomial has no more than one root in the range $(0, 1)$. For this reason, that is, for the configuration of the real roots of p, the work is likely to be unbalanced among the recursive calls in Algorithm 2. Consequently, it is likely to be unbalanced among the two calls to RootsInZeroOne in Algorithm 1 as well. Therefore, one should look for other opportunities for parallel execution.

The only other possibility is the Taylor shift computations. There are clearly several ways of "filling the Pascal's triangle". Setting the origin $(0, 0)$ at the top left corner, one may proceed row by row, or column per column, or with the elements of coordinates (i, j) satisfying $i + j = k$ for k successively equal to $= 0, 1, \ldots, d$. We call this latter traversal *anti-diagonal*.

This observation suggests a potential for concurrency. As mentioned in the introduction, we discuss two parallelization schemes: one is a *divide-and-conquer* approach (d-n-c, for short) and the other is a *blocking strategy* (blocking, for short). They are discussed in detail in Sections 4 and 5, respectively.

4. A divide-and-conquer strategy for computing the Taylor shift in parallel

The elements of the Pascal's triangle, and thus the coefficients of $f(x + 1)$, can be computed in a *divide-and-conquer* manner, demonstrated by the Subfigures (a) and (b) in Figure 1. Each triangular or square region is recursively divided into smaller regions until a base case is reached. Clearly, in both the triangular division and the square (or two-way tableau) division, opportunities for parallel execution are created. To be precise, the regions labeled by II can be computed concurrently.

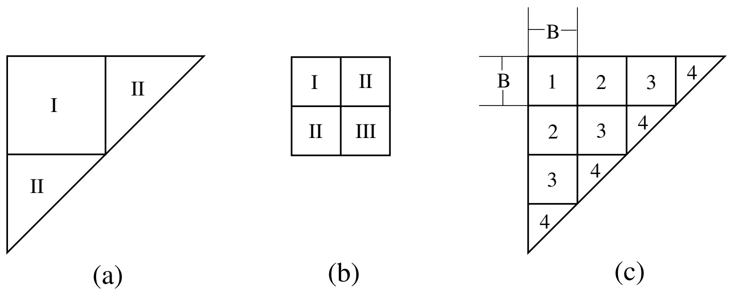


Figure 1. Illustration for the d-n-c and blocking strategies

In the rest of this paper, we assume that adding two integers is done within a constant amount of time. This property is true for machine integers. For multi-precision integers, this assumption is necessary in order to keep the analysis of the parallelization of Algorithms 1 and Algorithm 2 simple. For large input polynomials, relaxing this assumption is more realistic and estimating the cost of each addition in the construction of the Pascal's triangle lead to finer results and more opportunities for parallelism. The level of technical details, however, increase substantially and we will report this study in a forthcoming paper.

Since adding two integers depends linearly on their size, assuming that each integer addition is done in constant time implies that each integer fits within a constant number of bits. Let C be this number. Based on this assumption, we analyze the d-n-c approach for different complexity measures relevant to a multicore implementation, with an emphasis on cache complexity.

4.1. Work, span and parallelism estimates.

Our estimates follow the fork-join multithreaded programming model of the Cilk language [7]. For a two-way tableau with n as input data size, the work $U_1(n)$ required for filling this tableau satisfies $U_1(n) = 4U_1(n/2) + \Theta(1)$, for $n > 1$, and $U_1(1) \in \Theta(1)$, which implies $U_1(n) \in \Theta(n^2)$. (For simplicity, we may think of n as a power of 2.) For the same construction, the span (or critical path, or depth) satisfies $U_\infty(n) = 3U_\infty(n/2) + \Theta(1)$, for $n > 1$, and $U_\infty(1) \in \Theta(1)$, which leads to $U_\infty(n) \in \Theta(n^{log_2 3})$.

For a Pascal's triangle with n as input data size, the work $T_1(n)$ satisfies $T_1(n) = 2T_1(n/2) + U_1(n/2)$, for $n > 1$, and $T_1(1) \in \Theta(1)$, which gives $T_1(n) \in \Theta(n^2)$. For the same construction, the span $T_\infty(n)$ satisfies $T_\infty(n) = T_\infty(n/2) + U_\infty(n/2)$, for $n > 1$, and $T_\infty(1) \in \Theta(1)$, which implies that $T_\infty(n) = \Theta(n^{log_2 3})$. Therefore, for both constructions, the parallelism is $\Theta(n^{2-log_2 3})$. Since $2 - log_2 3$ is approximately 0.45, this indicates a relatively low parallelism, unless n is very large.

4.2. 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 input $2n$ integers. At completion, the output n integers can be found in this allocated space, most likely in its first n slots.

The same property is easy to prove for the d-n-c approach. To do so, one first establishes by induction a similar property for the two-way tableau construction, see Subfigure (b) in Figure 1. Then, one proves by induction the desired property for the divide-and-conquer approach of the Pascal's triangle depicted by Subfigure (a) in Figure 1.

Finally, we conclude that the d-n-c approach can be executed in place in space $2\,n\,C$.

4.3. Cache complexity estimates.

We use the (Z, L) ideal cache model of [6], where Z is the number of words in the cache and L is the number of words per cache line. Let $Q_S(n)$ (resp. $Q_T(n)$) be the number of cache misses incurred by the d-n-c approach at the level of the two-way tableau (resp. Pascal's triangle) on input data of size n.

When n is small enough, all the data read and written by the algorithm fit in cache. In this case, the number of cache misses is proportional to the amount of read/written data. Thus, there exists a positive constant α such that for $n \leq \alpha Z$ we have $Q_S(n) = 2n/L + 2$. Here we use the fact that the whole algorithm is executed in place within $2n$ integer slots. The $+2$ in $2n/L + 2$ comes from the fact that scanning an array of m consecutive words in memory incurs m/L or $m/L + 1$, depending on alignment issues.

For $n > \alpha Z$, we have the obvious equality $Q_S(n) = 4Q_S(n/2) + \Theta(1)$. To summarize, we have:

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

Solving this recurrence equation, we obtain:

$$Q_S(n) = \Theta(n^2/(ZL)).$$

The cache complexity analysis of the d-n-c construction of the Pascal's triangle leads to a similar recurrence equation:

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

for a positive constant α which can be chosen equal to the previous one. Elementary calculations leads finally to:

$$Q_T(n) = \Theta(n^2/(ZL)). \tag{1}$$

Next we prove that such a cache complexity estimate result is optimal. Since the cache complexity of the Pascal's triangle is dominated by that of the Tableau construction, it is enough to show that $Q_S(n)$ is optimal. To this end, we rely on the lower bound of the I/O complexity on the product of two direct lines established by Hong and Kung [8]. As shown in Figure 7-1 of [8], a Tableau is exactly the product of two direct lines L_2. From Section 8 of [8], we have $Q = |V|/S$, where Q is the minimum I/O time, V is the sequential time and S is the number of red pebbles available. Translating those into the language of cache complexity, we have $|V| = n^2$, $S = Z$ and $Q/L = n^2/(ZL)$ is the minimum number of cache misses. Thus $Q_S(n)$ is cache optimal.

One may wonder how things could be worse. In fact, it is not hard to verify that, for the two-way Tableau a row-by-row construction or a column-by-column construction would lead to a cache complexity of $\Theta(n^2/L + n)$. Since, in practice, Z is quite large (typically a few Megabytes for a L2 cache), the result of Formula (1) is much more attractive.

5. A blocking strategy for computing the Taylor shift in parallel

As noted in Section 4, the d-n-c suffers from a relatively low parallelism. There is, in fact, a hint for increasing the parallelism. One can observe that, in the above triangle division, parts of the small triangle regions II can be evaluated before completing region I.

A first solution implementing this observation is to increase the number of "ways" in the above divide-and-conquer algorithm. However, this leads to very complex code. In addition, this increases the cache complexity, A better solution is to adopt a *blocking strategy*. This is achieved by partitioning the entire Pascal's triangle into blocks of format $B \times B$, as shown in Subfigure (c) of Figure 1. Of course B should be tuned in order for a block to fit in cache. Then, the blocks are visited (and their elements are computed) following an anti-diagonal traversal.

5.1. Work, span and parallelism estimates.

The work for each block is $\Theta(B^2)$. Since we have $(n/B)^2$ blocks, we retrieve the work $T_1(n) \in \Theta(n^2)$. The span for each block is $\Theta(B^2)$. Indeed, each block is executed serially. Since there are n/B parallel steps in the anti-diagonal traversal, the span of the whole algorithm is $\Theta(Bn)$. So the parallelism is $\Theta(n^2/(Bn))$, which is $\Theta(n/B)$. This result is more attractive than the parallelism of the d-n-c approach. However, the d-n-c approach has the advantage of being cache-oblivious, that is, the knowledge of the characteristics of the cache is not required in order to minimize the number of cache misses in practice.

5.2. Space complexity estimate.

For each $B \times B$ block, the computation is sequential and is easily carried out in place within $2B$ integer slots. At the k-th parallel step k blocks are being computing while $2n/B - 2k$ ranges of B input integers have not been used yet. Therefore, in total $(2n/B - 2k)B + k\,2B = 2n$ integer slots are sufficient to run the whole algorithm.

5.3. Cache complexity estimates.

Let α be the constant introduced in the cache complexity analysis of the d-n-c approach. Assume that $B = \alpha Z$ holds. 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 $B = \alpha Z$ holds, we retrieve the optimal cache complexity result established for the d-n-c approach.

6. Optimizing the multicore implementation

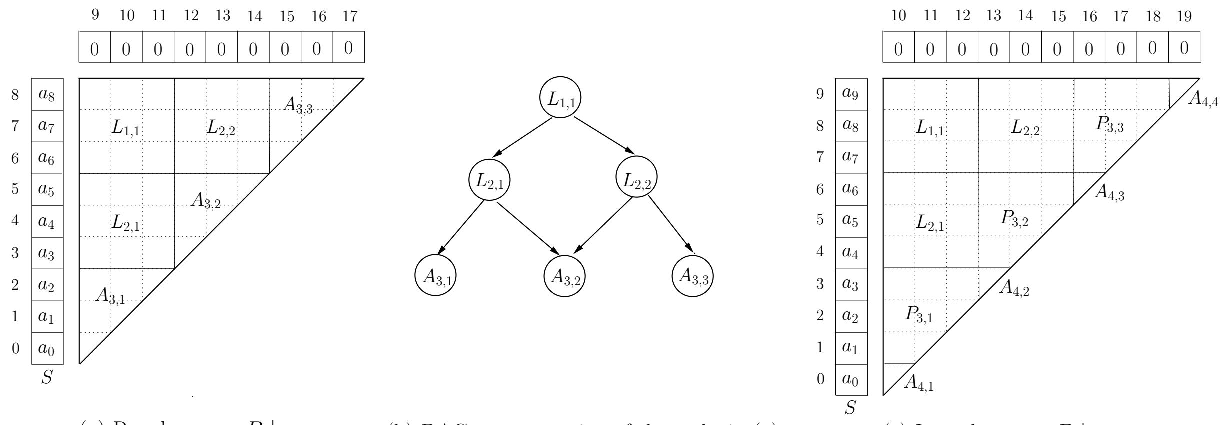
In this section, in Section 7 and in Section 8, we report on our implementation techniques for parallelization of Taylor shift computations targeting multicores. We show how to implement the d-n-c approach and the blocking strategy such that computing the Taylor shift of a univariate polynomial of degree d runs in-place using space $2Cn$ where $n = d + 1$. As before, C is the number of bytes for encoding a coefficient type (32-bit integer, 64-bit long integer or GMP multi-precision integer of bounded size). Recall that parallel execution can be achieved following either the blocking strategy or the the d-n-c approach, described respectively in Section 4 and Section 5. For both, the number of additions, and thus the work, is the same as that in the sequential construction of a Pascal triangle. However, the blocking strategy is theoretically more attractive in terms of parallelism whereas the d-n-c approach is cache-oblivious.

Given a polynomial of degree d, the number of blocks in the blocking strategy depends on the block size. Similarly, the number of division steps (or recursions) in the d-n-c approach depends on the *base case*. The best size of the base case shall be the one by which the program gives the best timing and speedup ratio, as a successful combination of many aspects, such as sufficient parallelism, low overhead of parallel constructs and function calls, low rate of cache misses, etc. In particular, the proper base case size may vary for polynomials with different coefficient size and on different computer architectures. Therefore, we include the size of the base case in our program as an adjustable (i.e. tunable) parameter. A user can first estimate the size of the base case based on the size of the coefficients of the input polynomial and the targeting computer architecture, and further fine-tune it by trial runs.

In the following two sections we describe our methods for in-place Taylor shift computation and for dynamically scheduling the tasks for concurrent execution according to the degree d of the input polynomial and the parameter B for the base case size. Section 7 covers the blocking strategy in detail while Section 8 offers an overview of the d-n-c strategy.

7. Optimizing the multicore implementation for the blocking strategy

Let $B > 0$ be the block size. For a polynomial f of degree d and $n = d+1 > B$, let q and r be the quotient and the remainder in the integer division of n by B. Thus we have $n = qB + r$, where $q \geq 1, r \geq 0$ and $r < B$. Depending on whether r is null or not, we classify the results of the blocking strategy into two cases: the *regular case* and *irregular case* corresponding respectively to $r = 0$ and $r > 0$. We illustrate them with examples in Figure 2.



(a) Regular case: $B \mid n$ (b) DAG representation of the tasks in (a) (c) Irregular case: $B \nmid n$

Figure 2. Case illustration for implementing the blocking strategy

7.1. The regular case: $B \mid n$

When $r = 0$, the entire computations in the Pascal triangle (let's name it PTR) are divided into small tableau of dimension $B \times B$ (for the intermediate computations) and small Pascal triangles of leg length B along the hypotenuse of PTR (near the end of the computation). For the example in Subfigure (a) of Figure 2 for which $d = 8$, $B = 3$ and hence $B \mid (n = 9)$, the blocking strategy divides the computations into three tableau of dimension 3×3 including $L_{1,1}$ on the first row and $L_{2,1}$ and $L_{2,2}$ on the second row, and three small Pascal triangles with leg length equal to 3 denoted by $A_{3,1}$, $A_{3,2}$ and $A_{3,3}$ on the last row.

In general we can deduce that there are $q-1$ rows of tableau of dimension $B \times B$ starting from the point of the right angle of PTR with each row parallel to the hypotenuse of PTR and only one row of small Pascal triangles of leg length B along the hypotenuse of PTR. Furthermore, if we index the $q-1$ rows of the small tableau by i where $i = 1, \cdots, q-1$, in the ith row there will be i number of tableaus with each dimension of $B \times B$. The number of small Pascal triangles along the hypotenuse of PTR is q.

Let us call the computation involved in each small tableau or a small Pascal triangle a *task*. The questions at this point are: How to schedule the tasks defined by the blocking strategy for parallel execution? What data structure can be used to minimize the synchronization cost for preserving the data coherency? How to store the intermediate data to obtain good locality?

According to the computational dependencies in the Taylor shift operation as shown in Example 1, the execution order of the tasks defined by the blocking method are deterministic. A task with indices (i, j) can not proceed until the one with indices $(i, j-1)$ and the one with indices $(i-1, j)$ are completed. In general, it can be represented by a Direct Acyclic Graph (DAG). In Figure 2, Subfigure (b) is a DAG representation of the example in Subfigure (a). However, the general DAG scheduling problem has been shown NP complete. In this work we design a parallel scheduling algorithm which is efficient in practice for computing Taylor shift on multicores. The basic idea is that the tasks with indices (i, j) for $j = 1, \cdots, i$ are executed in parallel. When all of them are completed, the tasks with indices $(i+1, j)$ for $j = 1, \cdots, i+1$ can start and run in parallel as well. This scheme is applied to all the tasks with indices (i, j) for $i = 1, \cdots, q$. In this way we exploit the parallelism offered by the many tasks on the same computational step while respecting the computational dependencies between two consecutive steps. The mapping between the tasks and CPUs are handled by the parallel platform Cilk++.

The entire Taylor shift operation is carried out in-place in an array S of size $2n$ with indices from 0 to $2n-1$. Initially the coefficients $a_0, a_1, \ldots, a_d$ of the input polynomial of degree d are copied to the first half space of S. The second half space of S are initialized to 0. An example is shown in Subfigure (a) of Figure 2. The working space of a task in S is scheduled so that it will not interleave with that of other tasks in parallel, to avoid from using locks for concurrent accessing to the same region of S. This is realized by properly arranging the space of S for storing the intermediate and the final results of the tasks in one parallel step, and its final results will be used as the input for some of the tasks in the next step.

More precisely, the working space of a task with indices (i, j), denoted by $S_{i,j}$, is the consecutive space between $S[n + (2j - i - 2)B]$ and $S[n + (2j - i)B]$ inclusively. Before the task starts, $S_{i,j}$ holds the input data (the results from some tasks in the $(i-1)$th step). The first half space of $S_{i,j}$ (with length B) holds the input data as the left side of a tableau or a triangle, while the second half holds the input data as the top side. The data accesses involved in the computation are carefully indexed so that all the computation can be done in-place. Further more, the final results are stored in the same space and are in the right order as the input for some of the tasks in the next step of computation. This way of memory management provides good data locality.

In the regular case, the in-place task processing is achieved by two procedures for the two different types of tasks, tableau and triangle, namely, TableauBaseInPlace(S, n, i, j, B), a sequential procedure to perform the computation for a task with indices (i, j) in the form of a tableau in work space $S_{i,j}$ and TriangleBaseInPlace(S, n, i, j, B), a sequential procedure to perform the computation for a task with indices (i, j) in the form of a Pascal triangle in work space $S_{i,j}$. The pseudo-code snippets for the two procedures are presented in Figure 3.

7.2. The irregular case: $B \nmid n$

An example of a case where $r > 0$ is given in Subfigure (b) of Figure 2. In the irregular case, we obtain also q rows of tasks. The tasks on the rows from 1 to $q-2$ are tableau with dimension of $B \times B$ and the tasks on the qth row (last row) are small Pascal triangles. However, the tasks on the $(q-1)$th row are neither tableau nor triangular. We call these tasks "polygon". It turns out that the scheduling algorithm for the regular case still works for the irregular case. We can also use a work space of size $2n$ to achieve in-place parallel computation. We only need to treat the "polygon" tasks by a special procedure for in-place processing, what we call PolygonBaseInPlace(S, n, i, j, B). Its pseudo-code snippet is shown in Figure 3.

```
TableauBaseInPlace($S, n, i, j, B$){
  $k = n + (2j - i - 2)B$
  for $u = B, \cdots, 2B - 1$ do
    $S[k + u] += S[k + u - 1]$
  for $u = B - 2, \cdots, 1$ do
    for $v = 0, \cdots, B - 1$ do
      $S[k + u + v + 1] = S[k + u + v]$
                          $+ S[k + u + v + 2]$
  $S[k] += S[k + 2]$
  for $u = 1, \cdots, B - 1$ do
    $S[k + u] = S[k + u - 1]$
                $+ S[k + u + 2]$
  $S[k + B] = S[k + B - 1]$
}
```

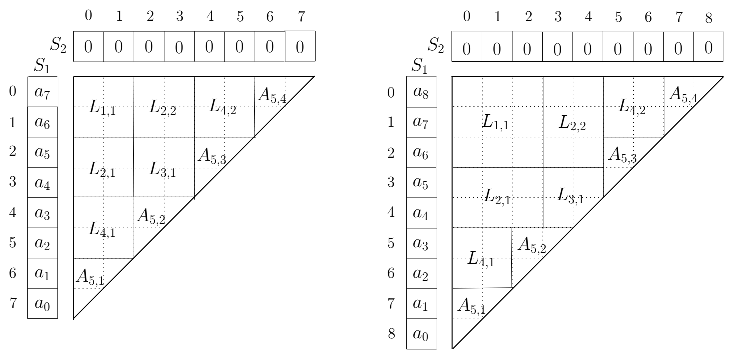
```
TriangleBaseInPlace($S, n, i, j, B$){
  $k = n + (2j - i - 2)B$
  for $u = 0, \cdots, B - 1$ do
    $S[k + B - 1] = S[k + B - 1]$
                    $+ S[k + B + u]$
    for $v = B - 2, \cdots, u$ do
      $S[k + v] = S[k + v] + S[k + v + 1]$
}
```

```
PolygonBaseInPlace($S, n, i, j, B$){
  $k = n + (2j - i - 2)B$
  $r = n \mod B$;  $t = B - r - 1$
  for $u = B, \cdots, 2B - 1$ do
    $S[k + u] += S[k + u - 1]$
  for $u = B - 2, \cdots, t$ do
    for $v = 0, \cdots, B - 1$ do
      $S[k + u + v + 1] = S[k + u + v]$
                          $+ S[k + u + v + 2]$
  for $u = t - 1, \cdots, 1$ do
    $S[k + t + 1] += S[k + u]$
    for $v = t + 2, \cdots, B + u$ do
      $S[k + v] += S[k + v - 1]$
  $S[k] += S[k + t + 1]$
  for $u = 1, \cdots, r$ do
    $S[k + u] = S[k + u - 1]$
                $+ S[k + u + t + 1]$
}
```

Figure 3. Key sub-procedures for implementing the blocking strategy

8. Optimizing the multicore implementation for the d-n-c strategy

In section 4, we have described briefly the d-n-c algorithm. Here we provide some implementation details.



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

Figure 4. Case illustration for implementing the d-n-c strategy

When n is a power of 2, as shown in (a) of Figure 4, each block in the divided Pascal's triangle is either a square tableau or a triangle. We call this case the regular case. When n is not a power of 2, as shown in (b) of Figure 4, a tableau may not be square. Indeed, the lengths of two edges of a tableau may differ by one. This irregular case suggests us to implement the divide-and-conquer approach using two arrays of n elements rather than one array with $2n$ elements as for the blocking strategy.

For efficiency reason, there is usually a sequential base case for a parallel divide-and-conquer algorithm. In our implementation of d-n-c algorithm, there are two types of base cases: tableau base case and triangle base case. We provide here the in-place implementation of the tableau base case. The implementation of the triangle case is similar.

Algorithm 3: TableauBase(p, s, q, t)

```
Input: a base block whose left edge is an array $p$ of length $s$
       and whose top edge is an array $q$ of length $t$
Output: the right edge of the block is stored in $p$ and the
        bottom edge of the block is stored in $q$
1 begin
2     for $i = 0$ to $t - 1$ do
3         $p[0] \leftarrow p[0] + q[i]$;
4         for $j = 1$ to $s - 1$ do
5             $p[j] \leftarrow p[j] + p[j - 1]$;
6         $q[i] \leftarrow p[s - 1]$;
7 end
```

9. Experimentation

We have implemented in `Cilk++` [9] the algorithms presented in Sections 4 and 5 together with the corresponding parallel version of the VCA Algorithm for real root isolation. For integer arithmetic, we rely on the `GMP` library [1].

We have tested our software on well-known benchmark examples, including the Bnd and Cnd polynomials from [11], the Chebychev and Mignotte polynomials from [5], as well as randomly generated polynomials. All the benchmark runs are carried out on an 8-core machine with 8 GB memory and 6 MB of L2 cache. Each processor is an Intel Xeon X5460 @3.16 GHz.

The timing results for a set of test cases with different configurations are summarized in Tables 1, 2 and 3. In each case, $n - 1$ and k denote the degree and the coefficient size (number of bits) of the input polynomial whereas B denotes both the *base case size* of d-n-c and the *block size* of blocking. The values of B reported herein are the ones which give the best timing after tuning.

The benchmark results show that our program reaches linear speedup on an 8-core machine for sufficiently large problems. We have also tested several other data traversal approaches for computing Taylor shifts and we have observed that our d-n-c and blocking outperform all of them. Overall, blocking demonstrates a slight win over d-n-c except for the case of Mignotte polynomials.

Table 1. Timings of ParTaylorShift (in seconds).

n	k	B	method	Bnd polynomial [11]			Cnd polynomial [11]			Random polynomial		
				8-core	1-core	Sp	8-core	1-core	Sp	8-core	1-core	Sp
5000	5000	50	blocking	1.33	6.49	4.9	0.92	2.26	2.5	1.33	6.50	4.9
5000	5000	8	d-n-c	1.45	6.63	4.6	0.94	2.33	2.5	1.45	6.63	4.6
10000	10000	50	blocking	7.70	50.76	6.6	4.40	17.46	4.0	7.79	50.78	6.5
10000	10000	8	d-n-c	8.54	51.66	6.0	4.22	17.64	4.2	8.51	51.65	6.1
25000	25000	50	blocking	103.9	778.9	7.5	42.6	261.2	6.1	103.9	778.7	7.5
25000	25000	8	d-n-c	110.1	789.6	7.2	41.5	261.7	6.3	110.2	789.7	7.2

In addition, we have used our software to isolate the real roots of a large polynomial coming from the study of limit cycles of dynamical systems on a 32-core machine. 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`. The 32-core machine has 8 Quad Core AMD

Opteron 8354 @2.2 GHz connected by 8 sockets; each core has 64 KB L1 data cache and 512 KB L2 cache; every four cores share 2 MB of L3 cache; the total memory is 126 GB. In about 15 minutes, our software manages to isolate all the 78 real roots of this large polynomial using this 32-core machine. However, we could not process this polynomial on a standard desktop (Intel Core 2 Quad CPU @2.40 GHz and 3.0 GB memory).

Table 2. Timings of Parris for Chebychev and Mignotte polynomials (in seconds).

n	B	method	Chebychev polynomial [5]				Mignotte polynomial [5]			
			8-core	4-core	1-core	Sp	8-core	4-core	1-core	Sp
400	50	blocking	59.09	106.41	413.87	7.0	166.69	222.65	564.91	3.4
400	8	d-n-c	59.02	107.55	420.18	7.1	128.57	181.95	572.65	4.5
500	50	blocking	173.83	323.18	1269.61	7.3	not enough			
500	8	d-n-c	173.92	324.48	1279.05	7.4	memory			

Table 3. Timings of Parris for random polynomials (in seconds).

n	k	d-n-c				blocking			
		B	8-core	1-core	Speedup	B	8-core	1-core	Speedup
1000	1000	8	0.94	3.26	3.5	50	0.87	3.21	3.7
2000	2000	8	3.51	18.84	5.4	50	3.25	18.58	5.7
3000	3000	8	4.07	23.33	5.7	50	3.82	22.89	6.0
4000	4000	8	38.26	246.34	6.4	50	35.84	243.82	6.8
5000	5000	8	200.62	1372.70	6.8	50	184.07	1340.95	7.3

10. Concluding remarks

In this paper, we revisit the core routine of many univariate real root isolation algorithms, namely the Taylor shift operation. We discuss two strategies (divide-and-conquer and blocking) to implement this operation in parallel on multicores. Assuming that the cost of adding two integers is constant, we show that both strategies are cache optimal. For the second approach, however, this requires to choose the block size accurately.

On an 8-core machine, for benchmark examples of relatively large size, the speedup for both the Taylor shift computation and the real root isolation is nearly linear. We also demonstrate the effectiveness of our software by processing a large real-world polynomial on a 32-core machine with more than a hundred of gigabytes memory.

The assumption that the cost of adding two integers is constant is considerably strong for the Taylor shift operation. Consider, indeed, a univariate polynomial $p(x)$ of degree d, with integer coefficients. Assume that the *height* of $p(x)$ (that is, the bit size of the maximum absolute value of a coefficients in $p(x)$) is bounded by h. Then, the height of $p(x+1)$ is bounded by $h+d$ [14]. Therefore, to further improve the work presented here, we are preparing a new report where our cache complexity analysis of the Taylor shift operation takes into account this potential increase of height. Since two blocks of the same size in the Pascal's triangle can be quite different in terms of the work required to fill them, we would like to design a new blocking strategy with *dynamical granularity* aiming at balancing the work load among the blocks.

References

[1] The GNU Multiple Precision Arithmetic Library. `http://gmplib.org`.

[2] C. Chen, M. Moreno Maza, and Y. Xie. Cache complexity and multicore implementation for univariate real root isolation. *ACM Commun. Comput. Algebra*, 44:97–98, January 2011.

[3] G. E. Collins, J. R. Johnson, and W. Küchlin. Parallel real root isolation using the coefficient sign variation method. *Lecture Notes in Computer Science*, (584):71–87, 1992.

[4] George E. Collins and Alkiviadis G. Akritas. Polynomial real root isolation using Descartes'rule of signs. In *proceedings of ISSAC'76*, pages 272–275, Yorktown Heights NY, 1976.

[5] T. Decker and W. Krandick. Parallel real root isolation using the descartes method. In *HIPC*, pages 261–268, 1999.

[6] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, 1999.

[7] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *ACM SIGPLAN*, 1998.

[8] Jia-Wei Hong and H. T. Kung. I/O complexity: the red-blue pebbling game. In *STOC*, pages 326–333, 1981.

[9] Intel Corporation. Cilk++. `http://www.cilk.com`.

[10] J. R. Johnson, W. Krandick, K. Lynch, D. G. Richardson, and A. D. Ruslanov. High-performance implementations of the descartes method. In *ISSAC '06*, pages 154–161, New York, NY, USA, 2006. ACM.

[11] J. R. Johnson, W. Krandick, and A. D. Ruslanov. Architecture-aware classical taylor shift by 1. In *ISSAC' 05*, pages 200–207, 2005.

[12] Maurice Mignotte. *Mathematics for computer algebra.* Springer, 1992.

[13] A. Strzebonski and E. Tsigaridas. Univariate real root isolation in an extension field. In *Proc. of ISSAC'11*, pages 321–328, New York, NY, USA, 2011. ACM.

[14] J. Von zur Gathen and J. Gerhard. Fast algorithms for taylor shifts and certain difference equations. In *ISSAC*, pages 40–47, 1997.