MATRIX FORM OF POLYNOMIAL REPRESENTATION ORIENTED TOWARDS FAST PARALLEL REAL ROOT ISOLATION

EUGENE V. ZIMA
Symbolic Computation Group
University of Waterloo
Waterloo, Ontario, Canada
ezima@daisy.uwaterloo.ca

ABSTRACT

The problem of parallelization of the well known family of algorithms to isolate polynomials' real roots is considered. Known complexity of sequential and parallel algorithms is $O(n^5 L(nd)^2)$ where n is the degree of the given polynomial, d is the maximum of absolute values of coefficients and $L(v)$ is the bit length of an arbitrary precision integer v. It is shown that with matrix form of polynomials' representation it is possible to achieve complexity $O(n^4 L(nd)^2)$. Combining this representation with some advanced sequential algorithm lets to derive a method of the complexity $O(n^3 L(nd))$.

KEYWORDS parallel processing, root isolation, fast shift.

1 INTRODUCTION

Polynomial root isolation is very important task for many applications ([10]). There exist a number of algorithms for solving this problem [6]. Most of them are based on coefficients sign variation method or Descartes' rule of signs. The best sequential complexity of these algorithms is $O(n^5 L(nd)^2)$, where n is the degree of the given polynomial, d is the maximum of absolute values of coefficients and $L(v)$ is the bit length of an arbitrary precision integer v.

Several approaches to the parallelization of real root isolation algorithms are described in [3, 9, 7]. However these approaches are based on multithreading and do not promise speed-up gained from parallelism bigger than n. In fact the complexity of the parallel algorithm proposed in [3] is not better than sequential one. In this paper we will consider an approach to the parallelization of the sign variation method to isolate real roots of polynomials. It can be easily combined with the approach from [3] and with techniques which in some steps of computations uses floating point arithmetic instead of exact one [7]. Our approach is based on the special matrix form of polynomial representation. Basic algorithms and representation proposed here are suitable for SIMD or array (including reconfigurable arrays) parallel architecture.

In section 2 we proceed with basic notions and definitions. In section 3 we consider the set of SIMD-like parallel operations needed to describe the solution of the problem. In section 4 we propose matrix form of polynomials representation and describe main operations for root isolation algorithms in terms of this representation together with complexity analysis (in the number of ring operations and in the number of bit operations). Section 5 summarizes theoretical results of the paper and describes possible future work.

2 PRELIMINARIES

Let

$$f(x) = a_n x^n + \ldots + a_1 x + a_0 \qquad (1)$$

be a univariate integral polynomial, I be an interval which contains some roots of the polynomial and we need to compute disjoint intervals in I such that each interval contains exactly one real root of (1). Denote $d = \max_{i=0,\ldots,n} |a_i|$, and $L(u)$ to be the bit-length of an integer u ($L(nm) \cong L(n)+L(m)$, $L(n^m) \cong mL(n)$). We assume throughout this paper that $f(x)$ has only single roots (i.e., that square free factorization procedure ([5]) has been already applied to the polynomial). Let

$$g(x) = b_n x^n + \ldots + b_1 x + b_0.$$

Algorithms to isolate real roots ([3, 9, 4]) are based on exact arithmetic and use the following three polynomials transformations as basics:

1. "coefficients scaling", $H_{1/2} : f(x) \rightarrow g(x)$, where $g(x) = 2^n f(x/2)$, i.e. $b_i = a_i 2^{n-i}, i = 0, 1, \ldots, n$.
2. "polynomial shift (translation) by 1", $T_1 : f(x) \rightarrow g(x)$, where $g(x) = f(x+1)$, i.e.

$$b_i = \sum_{k=i}^{n} a_k \binom{k}{i}, \quad i = 0, 1, \ldots, n. \qquad (2)$$

3. "inversion", $R : f(x) \to g(x)$, where $g(x) = x^n f(1/x)$, i.e. $b_i = a_{n-i}, i = 0, 1, \ldots, n$.

A trace of an algorithm to isolate real roots can be represented by binary tree [3, 7] where each node corresponds to a recursive call of the algorithm. With each node of the tree a polynomial and an interval are associated. The root of the tree corresponds to the initial polynomial $f(x)$ and interval I. If $f(x)$ has exactly one root on this interval (or does not have any roots at all), algorithms stops. Otherwise, transformations $H_{1/2}$ and T_1 are used in order to construct the children of the root with polynomials $H_{1/2}(f(x))$ and $T_1(f(x))$ and intervals I_1 (left half of I) and I_2 (right half of I). Then algorithm has to be applied recursively to these nodes. The decision procedure (which checks at each node how many roots correspondent polynomial has in correspondent interval) uses transformations T_1 and R. It looks on the number of sign variations in the sequence of coefficients of the polynomial $T_1(R(f(x))$. This means, that at each inner node at least two translations, one inverse and one scaling have to be performed.

It's easy to see that at each node the complexity of the translation dominates the complexity of any transformation. For example, if we count the complexity in the number of ring operations and consider sequential model of computations, we have $O(n^2)$ as the complexity of translation and $O(n)$ as the complexity of the inverse and scaling in the root node. If we count complexity as the number of bit operations, we have even bigger difference: e.g., $O(n^3+n^2L(d))$ is the complexity of the translation, $O(nL(d))$ is the complexity of the inverse.

In the case of parallel computations [3] the situation remains the same, since parallelization technique used there is based on the parallel performing of transformations on each level of the tree. Therefore, the speed-up obtained can not be higher then the average width of the tree, which is known to be less then n. Moreover, parallel complexity reported in [3, 7] is even the same as the best sequential complexity. It can be explained by the fact, that the Horner scheme of quadratic complexity to perform T_1 transformation is hard to parallelize because of data dependencies. That is why for example algorithm from [9] uses n processes to parallelize straightforward formula (2), which is of cubic complexity.

The complexity of the any algorithm considered here can be estimated from the trace tree described above. One of the factor implying complexity is the growth of coefficients size with the increase of the number of the level of the tree. If we start with polynomial $f(x)$ with certain value of $L(d)$, the length of the largest coefficient of polynomials at level l is dominated by $2nl + L(d)$.

Another factor implying the total comlpexity of the algorithm is the height of the trace tree, which is dominated by $nL(nd)$ [3]. Some algorithms (for example [4]) use transformation $T_c : f(x) \to g(x)$, where $g(x) = f(x+c), c \geq 1$ instead of T_1 on the stage of children construction. It often lets one to decrease the height of the tree. However the transformation itself is more complicated then T_1. Straightforward formula for coefficients of the polynomial $g(x) = T_c(f(x))$ looks like

$$b_i = \sum_{k=i}^{n} a_k \binom{k}{i} c^{k-i}, \quad i = 0, 1, \ldots, n.$$

The goal of this paper is to reduce the amount of work at each node of the tree with the help of the special form of polynomial representation. In this form it will be possible to exploit inner parallelizm of transformations T_1 and $H_{1/2}$ and to avoid performing of the transformation R at all. One of the main features of this representation is that algorithms to perform all transformations mentioned above are easy to implement. This representation requires some preliminary work before the start of real root isolation algorithm. However, the complexity of this work is reasonably small, and the work itself uses the same set of parallel tools as isolating algorithm. Bit-wise complexity of algorithm to isolate real roots of polynomials in this representation is $O(n^4L(nd)^2)$, or $O(n^3L(nd))$ if the technique from [7] is used. The complexity counted in the number of ring operations is $O(n \log nL(nd))$.

3 Basic SIMD operations

An usual dense representation of q-variate polynomial $f(x_1, \ldots, x_q)$ is q-dimensional array of coefficients. Considering basic SIMD-like operations on such arrays we will suppose that each entry of such an array is located in separate processor element (PE) and neighbors entries are located in neighbors PEs. Given a q-dimensional array $s[0..n_1, \ldots, 0..n_q]$ and $i \in \{i_1, \ldots, i_q\}$, we will use the following operations as basic:

- $\texttt{LeftShift}_i(s)$ shifts an array s one component to the left in the i direction (here "left" means towards decreasing i);
- $\texttt{RightShift}_i(s)$ shifts an array s one component to the right in the i direction (here "right" means towards increasing i);
- $s|_{i=j}$ denotes the $(q-1)$-dimensional sub-array of the array s obtained by fixing the value of the index $i = j$, where $0 \leq j \leq n_i$;
- $F(j)|_{i=j}$ denotes the q-dimensional array of the same shape as s, whose elements for $i = j$ and for any value of other indexes $i_1, \ldots, i_q$ are equal to $F(j)$.

Observe that every operation like shifting, computing $s|_{i=j}$ or e.g., $(j+1)|_{i=j}$, corresponds to a single parallel instruction on a SIMD machine and takes constant time [1] (of course under assumption that we do have enough PEs).

Additionally we consider binary parallel operations as basic. Let s and u be q-dimensional arrays of the same shape. Further we will use

- $s+u$ – component-wise addition of s and u;
- $s*u$ – component-wise multiplication of s and u;
- $u := s$ – component-wise assignment.

As usually for SIMD computations we assume that arrays of the same shape are mapped to the same set of PEs, i.e. entries $u[i_1,\ldots,i_q]$ and $s[i_1,\ldots,i_q]$ for fixed $i_1,\ldots,i_q$ are located in the same PE. That is why binary operations on these arrays take constant time.

Using operations above we can compose more complex expressions. For example,

$$u := (2^j)|_{i_2=j} * s \tag{3}$$

can be rewritten for explanation sequentially as

```
for all i ∈ {i_1,...,i_q}&i ≠ i_2  do
  for  j := 0,1,...,n_2  do
     u[i_1, j, i_3,...,i_q] := 2^j * s[i_1, j, i_3,...,i_q]
  od
od
```

This explanations contains loops, but parallel complexity of the assignment (3) is constant and consists of the following three steps:

1. temporary parallel variable of the same shape as s is assigned by values $(2^j)|_{i_2=j}$,
2. parallel multiplication of this variable and s,
3. parallel assignment of the result to u.

Let s be as before a q-dimensional array, u be a $(q-1)$-dimensional array, and $i \in \{i_1,\ldots,i_q\}$. A bit more complicated operations needed further are

- $\texttt{ReduceAdd}_i(s)$, which returns $(q-1)$-dimensional array
$$\sum_{j=0}^{n_i} s|_{i=j};$$
- $\texttt{CopySpread}_i(u)$, which returns q-dimensional array obtained by creating and spreading n_i+1 copies of u along axis i.

Both operations need $O(\log n_i)$ parallel steps ([2, 8]).

[1] We assume here that the complexity is counted in the number of ring operations.

4 Matrix representation of polynomials for real root isolation

Given a polynomial $f(x)$ of the form (1) we will use in the solution of the problem the following one and two-dimensional arrays:

1. vector $a[0..n]$ of coefficients of $f(x)$;
2. temporary matrix $A[0..n, 0..n]$ such that $A_{ij} = a_i$, $i = 0,1,\ldots,n; j = 0,1,\ldots n$; this matrix can be easily obtained from array a with the help of the parallel assignment $A := \texttt{CopySpread}_i(a)$;
3. vector $H[0..n]$ such that $H_i = 2^{n-i}$, $i = 0,1,\ldots,n$; this vector can be easily obtained with the help of parallel assignment $H := 2^{n-j}|_{i=j}$;
4.
   - matrix $T[0..n, 0..n]$ such that $T_{ij} = \binom{i}{j}$, $i = 0,1,\ldots,n; j = 0,1,\ldots n$ (we assume as usually that $\binom{i}{j} = 0$ for $i < j$); it is easy to see that this matrix represents Pascal's triangle and contains all the binomial coefficients from the formula (2).
   - matrix $T'[0..n, 0..n]$ such that $T'_{ij} = \binom{n-i}{j}$, $i = 0,1,\ldots,n; j = 0,1,\ldots n$ (the same Pascal's triangle but with inverse layout)

   Simple parallel algorithm to construct these two matrices is

```
T := 0; T' := 0; T[0,0] := 1; T'[n,0] := 1;
 for k := 1  to  n  do
       T|_{i=k} := T|_{i=k-1} + RightShift_j(T|_{i=k-1});
       T'|_{i=n-k} := T|_{i=k}
 od
```

Now we can define all transformations associated with each node of the trace tree in terms of these structures;

tr1) $H_{1/2}(f(x))$: $H*a$;
tr2) $T_1(f(x))$: $\texttt{ReduceAdd}_j(\texttt{CopySpread}_i(a)*T)$;
tr3) $T_1(R(f(x)))$: $\texttt{ReduceAdd}_j(\texttt{CopySpread}_i(a)*T')$.

Let's now estimate the amount of work at each node of the level l of the trace tree. The size (bit-length) of entries of a at the level l is dominated by $2nl + L(d)$. Entries of H are not changing during the algorithm run, and the size of entries of H is dominated by n. The same concerns to the size of entries of T and T'. Knowing this we can write down worst case bounds for complexity of transformations tr1-tr3. The complexity of tr1 is dominated by $(2nl + L(d))$ (because multiplication by 2^{n-i} can be performed as the bit-wise shift). Complexity of CopySpread in tr2 is dominated by $\log n(2nl+L(d))$, complexity of multiplication is dominated by $n(2nl + L(d))$ and complexity of `ReduceAdd` is dominated by $\log n(2n(l+1) + L(d))$. Since we avoid performing coefficients inverse in tr3, the same estimations take place for complexity of tr3. Summarizing all above (taking into account that $L(d)$ is a constant) we have the following estimation for the amount of work at the node on level l: $O(n^2 l)$. Hence, total complexity of the algorithm is

$$\sum_{l=0}^{nL(nd)} O(n^2 l) = O(n^4 L(nd)^2).$$

If we count complexity in the number of ring operations we have $O(1)$ for tr1, $O(\log n)$ for tr2-tr3 and complexity of algorithm is $O(n \log n L(nd))$.

5 CONCLUSION

The purpose of this paper is to show that it is possible to have speed-up of the algorithms to isolate real roots of polynomials gained from SIMD-like parallelism. We can proceed even better if we combine exact and floating point arithmetic as in [7]. An approach presented in [7] advises to use exact arithmetic for tr1-tr2 and floating point arithmetic for tr3. In [11] we described the representation of polynomials oriented towards fast parallel polynomial shift, which allows us to perform T_1 in $O(1)$ ring parallel operations ($O(n \log n)$ bit operations) if T_1 is applied to the polynomial repeatedly. Unfortunately, applying R to a polynomial destroys this representation, and we are loosing the speed gained because of reconstruction of this representation. However $H_{1/2}$ does not destroy this representation, which means that tr1-tr2 can be performed with complexity $O(n \log n)$ as in [11] and tr3 can be performed with complexity $O(n^2)$. It will lead to the total complexity of real root isolation $O(n^3 L(nd))$.

Possible future work in this direction includes some experimental implementation of real root isolation algorithms with representation described above and comparison with other implementations. It seems to be promising to get more advantages from parallelism on the level of implementation of arbitrary precision integer arithmetic, because applications which come from mechanics deal with polynomials of very high degree.

References

[1] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1989.

[2] *C* users guide.* Thinking Machine Co., 1992.

[3] G. Collins, J. Johnson, W. Küchlin. Parallel real root isolation using sign variation method. In R.Zippel, editor, *Computer Algebra and Parallelism*, pages 71–78. Springer Verlag, LNCS 584, 1992.

[4] G. Collins, A.Akritas Polynomial real root isolation using Descartes' rule of signs. In *ACM SSAC*, pages 272–275, 1976. ACM Press.

[5] J. Davenport, Y. Siret, E. Tournier. *Calcul formel.* Masson, 1987.

[6] J. Johnson. *Algorithms for Polynomial Real Root Isolation.* Technical research report OSU-CISRC-8/91-TR21 of the Ohio State University, 1991.

[7] J. Johnson., W. Krandick. Polynomial Real Root Isolation using Approximate Arithmetic. In *ISSAC'97*, pages 225–232, Maui, Hawaii, July 1997. ACM Press.

[8] W. Koch. Efficient Reduce and Scan Functions for Mesh-Connected SIMD Computers. In *PARCELLA'96*, pages 174–183, Berlin, Germany, 1996. Akademie Verlag.

[9] W. Krandick. Isolierung reeller nullstellen von polynomen. In J.Herzberger, editor, *Wissenschaftliches Rechnen*, pages 105–154. Akademie Verlag, Berlin, 1995.

[10] W. Krandick. Parallel Polynomial Real Root Isolation. 3rd International IMACS Conference on Applications of Computer Algebra, Maui, Hawaii, July 1997.

[11] E. Zima. Fast parallel computation of the polynomial shift. In *IPPS'97*, Geneva, Switzerland, April 1997. ACM Press.