Analysis and Modification of a Parallel Polynomial Root-finding Algorithm using Message Passing Interface

Parita Pooj, Priyanka Padmanabhan, Shweta Khusu and Sneha Kamath
Electrical Engineering Department, VJTI, Mumbai, India

Abstract—Traditional sequential algorithms are modified for parallel implementation in order to harness the computational potential of parallel platforms. This research paper discusses the implementation of an established polynomial root finding algorithm for a parallel platform and proposes modifications to it. Further, a technique to implement the algorithm on a multi-processor platform using Message Passing Interface (MPI) standard has been put forth.

Keywords—Parallel algorithms, Polynomial Root finding, Multicore Processing, Message Passing

I. INTRODUCTION
Parallelism plays an important role today on every platform from high-end industrial applications to low-end domestic ones. It is recognized that increased computational performance is not subsequent to faster processors, but parallel ones. This can be explained by the fact that processors clock cycle times are decreasing at a much slower pace, as physical restraints such as speed of light are approached. Therefore, the only way to increase the amount of operations that can be performed by computer architecture is by executing them in parallel [9].

In order to solve a problem efficiently on a parallel system, it is necessary to design an algorithm that specifies multiple operations on each step and takes into account the architecture and mode of parallel operation of the device along with the various overheads involved. In case of multi-processor systems with distributed as well as shared memory, networks of workstations etc., MPI is a useful paradigm for management and passing of resources in parallel computing [16]. MPI has replaced most other message-passing systems in technical computing. It has a high communication performance due to its rich set of communication features and is especially suited for homogeneous systems and multi-core platforms [17].

Traditionally sequential applications are being modified for parallel platforms for improved efficiency of the system [8] [10]. For instance, a numerical approach to replace steam & water properties of Steam Water Vapor Space (SWVS) with polynomial approximation functions suitable for simulations in cloud environment/multi-processor platform is described in [7]. Determination of all the roots of a high order polynomial is a classical problem of computational mathematics which finds application in many engineering sectors. Various root finding algorithms have been developed over the years [11] [15]. A wide research has been conducted on parallel root finding algorithms for various platforms and applications [5][8][12]. An NC algorithm that approximates all roots of polynomial having only real roots is given in [2]. Although unlike other methods then, it does not suffer from instability related issues, it is quite impractical [1]. A generalized version of [2], without the restriction of real roots is described in [3]. This algorithm too is impractical like its predecessor. A simpler NC algorithm on the same lines was implemented by Ben-Or and Tiwari [4]. A practical version of this algorithm suitable for implementation on parallel machines, with a fixed number of processors, is given in [1][18].

Almost all these algorithms primarily employ the bisection method on isolated root intervals to get the desired roots. Thus, effectively isolating the roots is an important aspect of these algorithms. The algorithm by Ben-Or [2] finds standard remainder sequence [14] of the given polynomial which is shown to depict properties of Sturm sequences [13]. Sturm sequences have inherent property that facilitates interleaving of polynomials of the sequence. This simplifies root isolation. Determination of the tree polynomials from the remainder sequence is carried out by a bottom-up transversal of the established tree nodes. This too adds to the scope of parallelism. In contrast to the robustness of this algorithm, the performance of some other algorithms of the time [5] is known to be sensitive to the choice of initial approximations of the roots, and the number of iterations needed for convergence has been shown to depend greatly and somewhat erratically on these choices [1].

The main contribution of this paper is to propose and develop a slight modification to the parallel implementation of Ben-Or and Tiwari’s root finding algorithm by Narendran and Tiwari in Section II. The algorithm is then designed for implementation on a multi-core platform using MPI standard between the cores in Section III. The simulation results are determined to test the efficiency of the algorithm in Section IV. This is followed by conclusion.
II. ROOT FINDING ALGORITHM

A. Root Isolation

This is a parallel algorithm to find simple, distinct roots of a polynomial of degree \( n \) with \( m \)-bit coefficients. First the standard remainder sequence \((F_0(x), F_1(x), \ldots, F_n(x))\) [14] of the given polynomial is calculated by using the extended Euclidean scheme for calculating polynomial GCDs [6].

\[
F_0(x) = p_0(x), \quad F_i(x) = p_0'(x)
\]

Where \( p_0(x) \) is the input polynomial, whose roots has to be found and \( p_0'(x) \) is the derivative of the polynomial

\[
F_1(x) = Q_1(x)F_0(x) - c_i^2F_i(x)
\]

\[
F_{i+1}(x) = Q_i(x)F_i(x) - c_i^2F_{i+1}(x)
\]

This standard remainder sequence satisfies the conditions of a Sturm sequence [13]. Through the Theorem 1 [18], a pair of polynomials, say \( p_i(x) \) and \( p_{i+1}(x) \), whose individual orders are roughly half of the original polynomial \( p_0(x) \) and sum to \( (n-1) \) can be computed. The roots of these two polynomials when collected and sorted \((y_{01}, y_{12}, \ldots)\) interleave the roots of the original polynomial \((x_0, x_1, x_2, \ldots)\); such that intervals for calculating the roots are obtained.

\[
x_0 \leq y_0 \leq x_1 \leq y_1 \leq x_2 \leq y_2 \leq x_3 \leq y_3 \leq \cdots
\]

For a polynomial \( p_i(x) \), we define \( k = \lfloor \frac{i + j + 1}{2} \rfloor \) such that for \( 1 \leq i \leq j \leq k \leq n \), the polynomials \( P_{i,k}(x) \) and \( P_{i+1,k}(x) \) are interleaving polynomials for \( P_i(x) \). A polynomial matrix is constructed by beginning with the original polynomial at node \((1,n)\) and using the value of \( k \) to recursively calculate the \( i, j \) values of the nodes of the polynomial tree until the leaf node is reached. The degree of any polynomial \( P_{i,j}(x) \) is given by \( j-i+1 \) except where \( i>j \). The leaf node denotes a polynomial of degree 1 whose root can be calculated directly. Due to the interleaving property, the roots of the polynomial of the children nodes when combined interleave the roots of the polynomial of the parent node.

![Fig.1 Polynomial tree for a polynomial with degree 8](image)

To facilitate the bottom-up transversal of the tree and to make computation easier, \( S \) and \( T \) matrices are defined [1][18] using the quotient sequence, \( Q_0, Q_1, \ldots Q_n \) and the leading coefficients of the polynomial. The \( T \) matrix at any parent node can be obtained from the \( T \) matrices of both the child nodes.

\[
P_{i,j}(x) = \begin{cases}
T_{i,j}[2,2] & i \leq j \leq n \\
F_{i,j}(x) & i \leq j \leq n ; j = n \\
1 & j \leq i
\end{cases}
\]

Where the matrices are defined as

\[
S_1 = \begin{pmatrix}
0 & 1 \\
-c_i^2 & Q_i(x)
\end{pmatrix}
\]

\[
S_i = \begin{pmatrix}
0 & 1 \\
-c_i^2 & Q(x)
\end{pmatrix}
\]

where \( 2 \leq i \leq n-1 \)

\[
T_{i,j} = \frac{T_{i+1,j}S_iT_{j,k+1}}{c_k^2 / c_{k-1}^2}
\]

When \( k = i \), \( P_{i+1,k}(x) \) strictly interleaves \( P_{i,j}(x) \). [1]
A. The Interval Problem

After the above computations, an approximate set of intervals encompassing one root each is obtained. The upper and
the lower limits of the roots of the polynomial are assumed to be associated with the number of bits of its roots. This can
be approximated to be the maximum number of bits of the polynomial’s coefficients, \( m \).

Therefore,

\[
y_0 = -2^m \quad \text{and} \quad y_n = 2^m
\]  

(9)

In every interval, the roots are calculated using certain approximation techniques. Therefore, \( \tilde{y}_0, \tilde{y}_1 \ldots \) are
the approximate values and not the absolute values of the intervals. A \( \mu \)-approximation technique is hence used to limit the
error such that the approximated value is greater than the actual value by a maximum error of \( 2^\mu \). \( \mu \)-approximation to the
root \( x_i \) is defined as

\[
\tilde{x}_i = 2^{\mu \|} 2^\mu x_i \\
\tilde{x}_i = 2^{\mu \|} (2^\mu x_i + 0.99\ldots) \\
\tilde{x}_i = x_i + 2^{\mu \|} \times 0.99\ldots
\]  

(10)

For max error \( \tilde{x}_i \approx x_i + 2^{\mu \|} \)

The interval obtained earlier needs to be modified to ensure that it is truly isolating and contains only a single root.
The following cases have to be considered [18]:

Case 1) If \( \tilde{y}_i = \tilde{y}_{i+1} \), then \( \tilde{x}_i = \tilde{y}_i \)

Case 2) If \( \tilde{y}_{i-1} - \tilde{y}_i \geq 2^{-\mu \|} \), then the root lies in one of three distinct regions:

Let \( r_i = i \) if \( \text{sign}(P_d(\pm\infty x)) = (-1)^i \text{sign}(P_d(\tilde{y}_i)) \)

\[
r_i = i + 1 \quad \text{otherwise}
\]

Here, \( r_i \) denotes the number of roots of \( P_d(x) \) in the interval \( (-\infty, \tilde{y}_i) \).

a) If \( r_i = i+1 \), the root lies in the interval \( (\tilde{y}_i - 2^\mu, \tilde{y}_i) \) and we can approximate \( \tilde{x}_i = \tilde{y}_i \).

b) We then calculate the value of \( P_d(x) \) at \( \tilde{y}_i \) and

\[
\tilde{y}_{i-1} - 2^\mu
\]

If \( \text{sign}(P_d(\tilde{y}_i)) \neq (-1)^i \text{sign}(P_d(\tilde{y}_{i-1} - 2^\mu)) \), a root lies in this interval and can be calculated using the bisection
method.

c) Else, the root lies in the interval \( (\tilde{y}_{i-1} - 2^\mu, \tilde{y}_{i+1}) \) and we can approximate \( \tilde{x}_i = \tilde{y}_{i+1} \).

The interval thus obtained is a truly isolating interval containing a single root of the polynomial. The approximation
technique utilized for estimating the root is the bisection method, where the limit on the number of bisections to be
performed is to be calculated. Let \((a, b)\) be the isolating interval containing the root. \( \log_2(b-a) \) number of bisections are
performed in order to obtain an interval of unit length. On the unit length interval, \( \mu \) bisections are performed such that
the size of the interval hence obtained is \( \frac{1}{2^\mu} = 2^\mu \). Since \( \mu \)-approximation technique was utilized, performing bisection
\( \log_2(b-a) + \mu \) times ensures that the obtained root is within the specified tolerance set by the value of \( \mu \). This is a
modification of the method employed by Narendran and Tiwari where a combination of Newton Raphson and Bisection
method was used to obtain the best approximation.

The above algorithm hence displays dual level parallelism. At a particular level in the tree, each node can compute its
\( T \) and \( P \) polynomials independently of the other nodes at its level. Also, for calculating the roots of a particular
polynomial each of the isolated intervals can work independently.

III. MPI IMPLEMENTATION

The message passing interface system is characterized by two major drawbacks - overheads due to passing excess data
and limitations in the number of parallel nodes, especially when implemented on a single machine. The algorithm
described in the previous section computes the \( \mu \)-approximation roots of a polynomial of degree \( n \) with distinct real roots
whose coefficients are of \( m \)-bit precision. The maximum number of processes needed for computation of roots efficiently
is given by \( 2^{[\log_2 n]} \). This is obtained by assuming the worst case scenario that the tree polynomial to be computed from
the remainder sequence is a full binary tree. If the number of processes specified is more than the number of physical
cores/processors, virtual nodes are created which share the CPU nodes using round robin scheduling [19][20]. We have
implemented the algorithm on a dual core processor which allows the use of a maximum of 2 physical nodes.

The program takes the polynomial as an input and other arguments such as \( \mu \), number of processes and \( m \) are passed to
the program. These parameters are used to compute the remainder sequence which can be later used to compute
necessary tree polynomials \( P_d \) using equations 5-8. The sequential implementation of this algorithm involves a recursive
function call to calculate the tree polynomials of each node using the polynomial obtained from the children. Thus the
algorithm follows a bottom up approach when computing the solutions of the tree polynomials starting with the leaves
(linear polynomials) and ending at the root node which is the polynomial whose roots we need.

Implementing this algorithm in parallel, a unique node to every polynomial on the same level in the tree polynomial is
assigned. The tree polynomials can be visualized as a binary tree where every node can be uniquely identified by \((i,j \ldots)\)
(leaf and level). Each node is assigned a process number based on its level and position in the binary tree, for instance Fig. 2. The
node assignment is based on the implementation of merge sort algorithm on MPI by Timothy Rolfe [21]. Based on this
paper, the mapping of nodes is done in a way that the parent node takes up the task of the left child instead of assigning a
new node for the left child, thus avoiding communication overhead between the child and the parent along with utilizing the idle time of parent.

Every node computes the remainder sequence which is a sequential task, so as to avoid overheads due to passing of the remainder sequence and the quotient sequence. In an alternate implementation, node 0 could compute the remainder sequence and pass the values down the tree to other nodes. However, this would increase the overheads due to passing huge arrays of data. The former method does not reduce parallelism since it utilizes its waiting time for computing the remainder sequence.

Any node can be uniquely identified by its rank and its level in the polynomial tree. A node may be performing either of two tasks at any given time - computing the tree polynomial for a given value of \( (i,j) \) (this value is different on different levels for the same process) or finding the root within an isolated interval, eg. Consider the node 0 - it computes the root polynomial by recursively calling the left child and creating new nodes for computing the right child at every level. At every node, after computing the tree polynomial from its children's \( T_{i,j} \) matrices, the node utilizes the same nodes for finding the roots from interleaving roots. \( n \) nodes are used for computing \( n \) roots by assigning each node an isolated interval. Thus, node 0 utilizes nodes 0,1,2,3 for a polynomial of degree 4. The nodes wait to receive the flag int variable which determines whether it should be computing a tree polynomial or a root.

A communicator determines the scope and the "communication universe" in which a point-to-point or collective operation between nodes is to operate [16]. Each communicator contains a group of valid participants. The source and destination of a message is identified by process rank within that group. Reusing the same nodes for computing a root within an isolated interval in parallel is done by initializing new groups and communicators by employing dual level parallelism. Computations at each level are synchronized by initializing non-overlapping communicators at every level between nodes that might have to compute the roots for the same polynomial.

The procedure is summarized as follows:
1. Calculate the maximum height of the binary tree for computing tree polynomials.
2. Calculate the remainder sequence at every node.
3. Initialize non-overlapping communicators at every node for each level.
   Eg. For \( n=7 \), level 1 initialises communicators between nodes [0 and 1], [2 and 3], [4 and 5], and [6 and 7] for ranks [0 and 1], [2 and 3], [4 and 5], and [6 and 7] respectively.
4. While true:
5. If rank = 0:
   - Roots = \( Tree\_poly (i, j, level) \)
   - \( Tree\_poly \) function calls the left child recursively and sends the flag int 0 and the \( i, j \) and \( next\_level \) values to the right child. It receives the computed \( T_{i,j} \) and the solutions of the left and the right child polynomials which interleave the polynomial at node \( i,j \). \( Tree\_poly \) then calculates its own tree polynomial. Once the interleaving roots are obtained:
     if \( i\neq j \), the root node sends flag int 1 to some nodes which may be needed to compute the root from isolated intervals. The new communicator corresponding to the level and the node is initialized between these nodes, which is later used to send the polynomial and to scatter the isolated intervals amongst the communicator nodes.
     If \( i=j \), compute the solution of the linear polynomial
       a. Send flag = 3 to all other processes in the communicator world to signal the end of algorithm.
6. else:
   a. Wait for the flag int

---

© 2014, IJARCSSE All Rights Reserved
i. flag int 0: Receive the values i, j and level from the parent for computing the tree polynomial. Call 
Tree_Poly function

ii. flag int 1: Compute the root from an isolated interval

Receive the root rank of the process whose polynomial’s roots are to be computed along with the
number of nodes that will be needed.
Initialize a new communicator between these nodes.
Calculate the root within an isolated interval provided by the parent root node using scatter()

iii. flag int 2: do nothing

iv. flag int 3: break

### TABLE I. SUMMARY OF RESULTS OBTAINED FROM SIMULATION FOR $\mu=8$

<table>
<thead>
<tr>
<th>Degree</th>
<th>No of Proc</th>
<th>Polynomial</th>
<th>Actual roots</th>
<th>Max. error of roots obtained from simulation</th>
<th>Average error of roots obtained from simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>1(x^2) + 3(x) + 2</td>
<td>[-2, -1]</td>
<td>1.14E-05</td>
<td>1.14E-05</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1.0(x^3) - 6.0(x^2) + 11.0(x) - 6.0</td>
<td>[1, 2, 3]</td>
<td>0.003875673</td>
<td>0.00193278</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1(x^4) - 5(x^3) + 4</td>
<td>[-2, -1, 1, 2]</td>
<td>0.003896066</td>
<td>0.001594537</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>1.0(x^8) - 14.1(x^7) + 77.75(x^6) - 177.5(x^5) + 56.5(x^4) - 11.0(x) + 0.0</td>
<td>[-2.5, -2, -1, 0, 1, 1.1, 2]</td>
<td>0.00313971</td>
<td>0.00122038</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>1.0(x^16) - 42.95(x^15) + 73.976(x^14) - 7.112(x^13) - 294.028(x^12) + 222.51(x^11) + 273.644(x^10) - 203.448(x^9) - 49.392(x) + 0.0</td>
<td>[-3.5, -2, -1, -0.2, 0, 1, 2, 2.1, 2.8, 3]</td>
<td>0.002081061</td>
<td>0.001104906</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>1.0(x^16) - 42.95(x^15) + 214.47(x^14) + 759.025(x^13) - 3910.82625(x^12) - 5617.76375(x^11) + 34332.0975(x^10) + 16074.645(x^9) - 149173.166(x^8) + 6681.64125(x^7) + 290320.245(x^6) - 94936.985(x^5) - 171778.32(x^4) + 77086.8(x) + 0.0</td>
<td>[-4, -3.5, -3, -2.3, -2, -1, 0.5, 1.2, 2.53, 3.5, 3.8, 4]</td>
<td>0.003437599</td>
<td>0.001579975</td>
</tr>
<tr>
<td>18</td>
<td>32</td>
<td>1.0(x^32) - 27(x^31) - 52.07(x^30) + 143.559(x^29) + 1060.2244(x^28) - 3020.52078(x^27) + 32166.535(x^26) + 55169.7023(x^25) - 184341.385(x^24) - 132231.178(x^23) + 55979.704(x^22) + 79299.0376(x^21) - 829887.076(x^20) + 165135.563(x^19) + 467009.764(x^18) - 188564.101(x^17) - 4587.66699(x) + 1869.04947</td>
<td>[-4, -3.6, -3.1, -2.7, -1.9, -1.5, -1.2, -0.1, 0.1, 0.5, 1.2, 1.5, 1.9, 2.2, 2.7, 3.1, 3.6, 4]</td>
<td>0.003572307</td>
<td>0.001328411</td>
</tr>
</tbody>
</table>

No of Proc = No. of virtual cores needed for the given degree polynomial

### IV. SIMULATION AND RESULTS

The simulation was carried out on Dell Inspiron 14R machine with dual core i3 processor. The code was written in Python language using MPI for Python package [22] with MPICH2 implementation. MPICH2 is an open source portable implementation of the MPI standard.

A summary of the results obtained from simulation is given in Table 1. The roots of arbitrarily chosen polynomials of different degrees were computed for a fixed value of $\mu=8$. For error given as observed root – actual root, the average error of the roots and the maximum error amongst them have been computed and mentioned. It is observed that both these errors are within the specified $\mu$-approximation tolerance ($2^{-\mu}$).

### V. CONCLUSIONS

The paper presented a modified implementation of Narendran and Tiwari’s root finding algorithm using Message Passing Interface standard. The observed roots were confirmed to be within $\mu$-approximation of the actual roots. The future work aims at analyzing the timing efficiency of the implementation. A single machine with dual cores may not be sufficient for optimum utilization of this technique. Hence, timing efficiency would be better analyzed on a larger number of cores or a cluster system. Further, the parallel root finding algorithm can be designed to be implemented on a GPU enabled device or FPGA development board.

© 2014, IJARCSSE All Rights Reserved
ACKNOWLEDGMENT

We would like to thank our project guide, Dr. M.S. Panse, Head of Department, Electrical Department, VJTI, for constant guidance and support during the course of this project.

REFERENCES