Some parallel methods for polynomial root-finding

A.J. MAEDER and S.A. WYNTON
Computer Science Department, Monash University, Melbourne, Australia

Received 10 July 1985

Abstract: Parallelizations of various different methods for determining the roots of a polynomial are discussed. These include methods which locate a single root only as well as those which find all roots. Some techniques for parallelizing such methods are identified and some examples are given. Further places in polynomial root-finding algorithms where parallel behaviour can be introduced are described. Results are presented for a range of programs written to test the effectiveness of methods presented here.

Keywords: Parallel algorithms, polynomial root-finding, zeros.

1. Introduction

Finding roots (or zeros) of a polynomial of degree \( n \) in \( x \), i.e. given

\[
P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0
\]

find one or more \( x_i \) such that \( P(x_i) = 0 \), \( 1 \leq x^i \leq n \), is a well known problem and many numerical methods of solution exist [7,19]. In general, both \( a_i \) and \( x_i \) are in \( \mathbb{C} \), but often problems are posed with \( a_i \) in \( \mathbb{R} \) only. The lack of a suitable closed formula for roots of polynomials with \( n \geq 5 \) encouraged the early development of simple iterative techniques. Originally intended for finding real \( x_i \) only, they were often equally useful for finding roots of any real function. These methods are surveyed widely in numerical literature and have inspired a number of more sophisticated algorithms in more recent times. This paper does not attempt to describe the range of known methods but instead will concentrate on their structure in an effort to identify properties suited to parallelization.

Two categories of root-finding algorithms may be distinguished which lend themselves to different forms of parallelism. These will be referred to as single-root and simultaneous-root algorithms. Many traditional algorithms converge to one root (or sometimes two) only and some further action is needed to find other roots. This may take the form of restarting the algorithm from some different initial conditions, e.g. an estimate for a different root. Alternatively the polynomial could be transformed in some way, e.g. deflating by the root previously found to obtain a polynomial of lower degree with the same unfound roots remaining. The former approach relies on sufficiently close approximations to other roots being available. The latter can result in compounding of roundoff error over successive deflations, possibly affecting conver-
gence to the desired root. There are also methods converging to all roots at the same time which can overcome the problems of choosing starting points and calculating roots with accumulated errors. Techniques for the parallelization of both single and simultaneous-root algorithms will be considered and results will be presented for parallel programs for some specific methods.

2. Single-root methods

A number of single root methods are based on successively approximating the polynomial until a zero of the approximating function is sufficiently close to a zero of \( P(x) \). The approximating function may be linear, as in the Secant and Newton–Raphson methods, or of higher order like Muller’s method. These methods are highly serial as the next approximation is calculated using the last approximation results each iteration step. However, by considering the work done within each step using data flow analysis techniques, independent parts of the calculation may be identified. Exploiting independence to produce parallel sections of code should reduce the time spent on each iteration step, although the algorithm still retains a synchronized behaviour. An example is the Newton–Raphson method where the \( k \)th iteration step consists of

\[
x(k) = x(k-1) - \frac{P(x(k-1))}{P'(x(k-1))}
\]

where the computations of \( P \) and \( P' \) may proceed independently once \( x(k-1) \) is available (see Fig. 1) [11]. A further time saving is made by checking the convergence criterion for the previous step, \(|P(x(k-1))| > E\), in parallel with the calculation of \( x(k) \). Some parts of the algorithm must be executed serially and some synchronizing mechanism is needed to indicate when \( x(k-1) \) is available so that the correct \( P \) and \( P' \) are obtained for the current step.

![Fig. 1. Parallel Newton–Raphson method.](image)
In the above example it was assumed that \( P'(x) \) was calculated by evaluating a polynomial so that the time taken was comparable with that needed to form \( P(x) \). The derivative may also be obtained by numerical differencing, as in the secant method

\[
x_{(k)} = x_{(k-1)} - \frac{P(x_{(k-1)}) - P(x_{(k-2)})}{x_{(k-1)} - x_{(k-2)}}
\]

where only four arithmetic operations are required to estimate the derivative. In this case \( P(x_{(k-1)}) \) must be calculated before the derivative can be estimated and only \( (x_{(k-1)} - x_{(k-2)}) \) can be performed in parallel with this calculation. This might result in wasted idle time on some parallel processors, unless they can be used instead to speed up the calculation of \( P(x_{(k-2)}) \) as will be discussed later. The more processors there are already allocated to the task of calculating \( P(x_{(k-1)}) \) the less effective these extra processors will be in reducing total computation time. If the maximum number of processors needed to calculate \( P(x_{(k-1)}) \) are already available, the extra processors will not shorten the computation time at all, so that in this case there would be little difference in total time taken between using the parallel secant and parallel Newton–Raphson methods.

The parallel organization suggested above for the Newton–Raphson method can be viewed as performing some preparation for the next iteration step (calculating \( P'(x_{(k-1)}) \)) at the same time as completing the current one (calculating \( P(x_{(k-1)}) \) to check for convergence). This overlapping or pipelining approach may be used to advantage in parallelizing other single-root methods, instead of the data flow approach. The advantage is that the algorithm need not be analysed at the level of individual arithmetic or machine instructions in order to derive the parallel version. Pipelining is feasible in Bairstow’s method (see Fig. 2), which finds a quadratic factor of the polynomial, thus detecting two roots (which may be real or complex). Assuming \( x^2 - u_0x - v_0 \) is close to a quadratic factor of \( P(x) \), form sequences \( \{u(k)\}, \{v(k)\} \) of improved estimates for the true factor by

\[
b_j = a_{n-j} + u(k)b_{j-1} + v(k)b_{j-2}, \quad 0 \leq j \leq n, \quad b_{-1} = b_{-2} = 0,
\]

\[
c_j = b_j + v(k)c_{j-1} + u(k)c_{j-2}, \quad 0 \leq j \leq n - 1, \quad c_{-1} = c_{-2} = 0.
\]

\[
u_{(k+1)} = u_{(k)} + (b_n c_{n-3} - b_{n-1} c_{n-2})/(c_{n-2}^2 - c_{n-1} c_{n-3}),
\]

\[
u_{(k+1)} = v_{(k)} + b_n (c_{n-1} - c_{n-2})/(c_{n-2}^2 - c_{n-1} c_{n-3}).
\]

The \( b_j \) depend only on previous \( b \) values, while the \( c_j \) depend on previous \( c \) values but also on the current \( b_j \). Thus two pipelines of calculations can be established, the \( b \) pipe consuming previous \( b \) values and producing new ones and the \( c \) pipe combining previous \( c \) values and either queueing these or including the corresponding \( b \) value if it is available. This allows some flexibility in how the parallel processes are run as either pipe can continue to perform useful work even though the other may be stopped. The necessary point of synchronization is after \( b_{n-1} \) has been calculated, when no further \( b \) pipe work can be done until the next \( u \) and \( v \) values have been found. If the pipes run at about the same rate, the amount of time taken should be approximately halved. The actual gains would be less than this due to startup delays and the cost of communicating the intermediate \( b \) pipe output values to the \( c \) pipe. A data flow examination of the work in each pipe reveals that the \( u_{(k)} \) and \( v_{(k)} \) multiplications can be performed simultaneously. This reduces the time taken for one item to be produced from the pipe to the
duration of three arithmetic operations (1 multiply, 2 adds) compared with the serial case of eight operations (4 multiplies, 4 adds).

An alternative to creating a new estimate for the zero by evaluating an approximating function at each iteration is to find a region enclosing the zero. A simple example of such bracketing or searching methods is the \textit{bisection method}, where the midpoint of an interval enclosing the zero is found by

\[ x_{(k)} = \frac{1}{2}(x_{(\text{left})} + x_{(\text{right})}) \]

and the sub-interval containing the zero chosen as the interval to be used in the next iteration. The bisection method has been an obvious candidate for parallelization. A straightforward approach is to evaluate the function simultaneously on \( m \) processors at \( m \) points within the interval, resulting in \( m + 1 \) sub-intervals [17]. If the original interval contained only 1 distinct root, then either one of the \( m \) points satisfies the convergence criterion or else one of the sub-intervals brackets the root. This replication of effort is termed \textit{redundancy} because the values
of $P$ at many endpoints of the sub-intervals might have to be determined before the new bracketing interval is found. Once this interval has been established any remaining unfinished evaluations may be discarded. Results for the parallel bisection method have been published by Shedler [16] as well as extensions of the technique involving other simple single-root methods. Some improvements of the basic algorithm are known e.g. Miranker [12] reduces the search interval length further by considering bounds on the function value given by the derivative over that interval.

3. Simultaneous-root methods

If estimates for all distinct roots of a polynomial were available, a conventional single-root method could be used for each one and all these computations executed in parallel. A SIMD computer would be able to perform such an algorithm, advancing each iteration step for every root synchronously. For reasons of efficiency it may be better to use a MIMD system which would allow each single-root iteration to take place independently. Different convergence times for various roots could be tolerated and different single-root methods could be used for each root.

A number of parallelized simultaneous-root algorithms already exist, many of them developed comparatively recently. Two classes can be distinguished: those based on geometric (searching) techniques and others on algebraic techniques. Geometric techniques involve subdividing the real line or complex plane [6,3] into regions of which some are known to contain a distinct root. Those without zeros are rejected and the whole process repeated until the convergence criteria are met. The regions may be circular [18], rectangular [20,10] or square [14]. A major difficulty is establishing regions known to contain exactly one distinct root. Once this has been achieved, convergence is usually obtained easily. A fast single-root method is often used to complete the convergence to a particular root rapidly. Algebraic techniques include the Durand–Kerner method [1], modified Newton methods [5] and the Quotient-Difference algorithm [7, pp. 162–179]. A novel method due to Patrick [13] also warrants mention. Derivatives of $P$ are taken until a degree 1 or 2 polynomial is obtained. This can be solved directly and the zeros used as starting values to find roots of the preceding polynomial of two higher degree, continuing until the original polynomial is solved.

Many geometric methods utilise Sturm sequences [15] to find estimates for the initial regions containing roots. A Sturm sequence is a sequence of non-vanishing real-valued functions with the property that at any zero of one of the functions the adjacent functions are non-zero and opposite in sign. By examining the values of all the sequence functions at various points, the number of zeros of the original polynomial in each interval can be calculated. For $P$ the Sturm sequence $(f_m)$ may be formed by taking

$$f_1(x) = P(x), \quad f_2(x) = P'(x),$$
$$f_{j+1}(x) = Q_{j+1}(x) \times f_j(x) - f_{j+1}(x), \quad j = 2, \ldots, m - 1$$

with $f_m(x)$ dividing $f_{m-1}(x)$ exactly.

Once the polynomial divisions have been completed, the functions are evaluated at chosen points and the intervals containing functions identified by counting the sign changes in function values
at each point. Each interval containing zeros is subdivided to form another set of points for evaluation, obtaining another set of intervals and so on.

Various stages of this method are suited to different forms of parallelization. Firstly, the process of dividing the polynomials to form each Sturm sequence function can be parallelized by finding each coefficient of the remainder directly from the coefficients in the arguments, analytically. As soon as the first function is available, values at the chosen points can be calculated in parallel using it. Once all the functions have been calculated and all the values at the chosen points obtained, the sign counting can proceed over all the points in parallel. At most $n$ intervals containing zeros can occur between these points. Each of the zero-containing intervals must be resubmitted for subdivision into a set of points followed by a Sturm sequence function evaluation phase to obtain a further refined set of intervals. There are thus two stages in the algorithm which may be parallelized. Each interval may have Sturm function values for its set of points computed in parallel or instead the splitting of intervals into sets of points may be carried out over all current intervals in parallel. The number of current intervals starts at one ($-\infty, \infty$) and increases slowly during execution of the program to $n$ finally. The number of points per interval would typically be constant. Depending on the number of processors available it may be possible to choose either or both of these parallel strategies.

The quotient-difference scheme [7], also known as Rutishauser’s method, is based on the calculation of sequences of quotients $q_k$ converging to the roots of $P$, alternating with the calculation of differences $e_k$ between successive steps. These are formed by

$$e_k^{(j)} = (q_k^{(j)} - q_k^{(j-1)}) + e_k^{(j-1)}, \quad q_k^{(j+1)} = \frac{e_k^{(j)}}{e_k^{(j)} q_k^{(j)+1}}.$$ 

with the initial $e_k^{(0)} = 0$, $q_k^{(1)} = \frac{b_k + 1}{b_k}, 1 \leq j \leq n - 1, k = 0, 1, 2 \ldots$,

where $b_k = -\frac{1}{a_n} (a_{n-1} b_{k-1} + a_{n-2} b_{k-2} + \cdots + a_0 b_{k-n})$.

Calculation of the successive $e$ and $q$ terms at each step is highly synchronous but speedup can be achieved by obtaining the next term of all sequences in parallel. This could equally well be done on a SIMD computer. However, an enhancement of the method is to speed up the normally slow convergence once the $q$ terms have started to stabilize by swapping to a fast single-root method and rapidly finishing the computation. Each of these iteration methods should be run independently as they will converge at different rates and thus a MIMD machine is desirable for this step at least. A complication arises in that a particular term which has been replaced by a single-root iteration may have neighbours which need its $q$ and $e$ values for the calculation of their own. This work could be carried out in parallel with the single-root iteration which is a task independent of successive $q$ and $e$ values.

4. Further opportunities for parallelization

Apart from computations making up a step in the particular method being used, there is scope for parallelizing some of the fundamental operations used by a step. As most root-finding
algorithms contain a number of polynomial evaluations, any savings in the work done here will produce savings in overall time. If calculated in the conventional way, \( P(x) \) requires \( O(n^2) \) arithmetic operations. Serial algorithms tend to use the more efficient Horner’s rule

\[
P(x_{(k)}) = \left((\cdots((a_n x_{(k)} + a_{n-1}) x_{(k)} + a_{n-2}) \cdots) x_{(k)} + a_0\right)
\]

which requires only \( O(n) \) operations. This calculation scheme is inherently sequential and consequently is no faster on a parallel machine. Dorn [4] generalizes Horner’s rule by forming polynomials summing to \( P \) which contain common multiples of powers of \( x \). Horner’s rule can now be applied to each small polynomial in parallel and the results combined, perhaps by cascading if many are involved. This involves pairing as many additions as possible for parallel evaluation, then pairing the results in the same way and so on until the final sum is found. The computation still uses \( O(n) \) operations if the number of processors remains fixed but obviously offers some savings. If an unlimited supply of processors is available, the cascading principle can be extended to compute the entire result using \( O(\log_2 n) \) operations [2].

An alternative formulation of Horner’s rule as a synthetic division process is sometimes used. In this form rows of successive multiplications by \( x_{(k)} \) and the associated additions are tabulated as intermediate results so the final answer is attained incrementally. This scheme has the advantage that it can be extended to evaluate all the derivatives of the polynomial by repeating the process, using the current row to form a new set of intermediate results which will produce the next derivative. An example is given by Henrici [7, pp. 55–56] (see Fig. 3). This procedure can be parallelized to some extent by computing each column in parallel with each row, since successive items in either depend only on values previously calculated. The procedure can also be used to obtain the coefficients of a deflated polynomial immediately from the table of values [7, pp. 84–86].

5. Experimental results

The results presented here were obtained by simulating processes running in parallel during execution of a serial program on a conventional SISD VAX 11-780 computer. Parallel process execution times were measured using operating system utilities which were separated checked for reliability and consistency. Where necessary, communication overheads for synchronization of data values between parallel processes were modelled by delays proportional to the size of the data concerned. The intention was to simulate storage time in shared random access memory. All programs were coded in high level languages (FORTRAN 77 and Pascal) and no attempt was made to optimise the code produced in any way. Since no claims about the accuracy of robustness of the test programs are made here, the particular polynomials used in testing will not be described in detail. In most cases they were based on design of test cases as presented by Jenkins and Traub [9]. Execution times have been normalised to the serial algorithm times except in cases where comparison between methods is being considered.

In the case of single-root methods, implementations of the parallel Newton–Raphson and secant methods as described in Section 2 were tested. In both cases a two processor algorithm was used. The secant method algorithm used the processor which was idle following the calculation of \( x_{(k-1)} - x_{(k-2)} \) to speed up the calculation of \( P(x_{(k-1)}) \) with maximum efficiency.
The results appear in Table 1. As expected, the Newton–Raphson speed-up was slightly better than that of the secant method.

Bairstow’s method was parallelized first by constructing $b$ and $c$ pipes but performing calculations serially within each pipe, i.e., a two processor algorithm. Then the algorithm was adapted to perform work within each pipe in parallel i.e. using four processors. The evaluation of $u_{(k+1)}$ and $v_{(k+1)}$ was carried out in parallel using all processors available in the particular algorithm. These results are shown in Table 2. The speed-ups obtained were reasonably close to the expected limits of around 2 and 2.67 for two and four processors respectively. These results
indicate that even on systems with a small number of processors, a good speed-up can be achieved using single-root iterative methods.

The parallel Sturm sequence method was implemented using a serial secant method for fast single-root convergence. Three versions of the Sturm program were used, corresponding to the different parallel behaviour possible as described in Section 3. S1 is the case where the Sturm function values for a single point in every interval are obtained in parallel. The number of processors needed is thus the maximum possible number of intervals i.e. \( n \), the degree of the P. S2 evaluates the Sturm functions at every point in an interval in parallel but processes only one interval at a time. The number of processors required here is the greater of \( n \) and the number of points used per interval (assumed constant). The third program S3 combines both of these parallel activities and evaluates all points of each interval in parallel and does so for all intervals in parallel, using \( (n \times \text{number of points per interval}) \) processors. In all three programs the number of points per interval was set at 20. As expected, the speed-ups for the three algorithms improve as the number of processors used increases i.e. as more parallelism is introduced into the algorithm. However, if the speed-up per processor is considered, S2 is clearly more efficient in its use of the parallel processors than the other two methods. These results are given in Table 3.

The quotient-difference algorithm tends to be fast as it does not have to perform polynomial evaluations at each step. Forming each \( q \) sequence or \( e \) sequence in parallel requires an \( n \)

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Parallel Newton–Raphson and secant method results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>Polynomial degree ( (n) )</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Newton–Raphson</td>
<td>30</td>
</tr>
<tr>
<td>Secant</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Parallel Bairstow method results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polynomial degree ( (n) )</td>
<td>Average serial time</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>30</td>
<td>1.00</td>
</tr>
<tr>
<td>50</td>
<td>1.71</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3</th>
<th>Parallel Sturm sequence method results</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>Serial time</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>5</td>
<td>1.000</td>
</tr>
<tr>
<td>10</td>
<td>5.199</td>
</tr>
<tr>
<td>15</td>
<td>22.30</td>
</tr>
<tr>
<td>20</td>
<td>43.79</td>
</tr>
</tbody>
</table>
Table 4
Parallel quotient-difference method results

<table>
<thead>
<tr>
<th>Polynomial degree ((n))</th>
<th>Average serial time</th>
<th>Average parallel time</th>
<th>Speed-up</th>
<th>Speed-up per proc</th>
<th>Jenkins–Traub average time</th>
<th>Speed-up over Jenkins–Traub</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1.0</td>
<td>0.60</td>
<td>1.67</td>
<td>0.33</td>
<td>2.2</td>
<td>3.7</td>
</tr>
<tr>
<td>10</td>
<td>4.5</td>
<td>1.55</td>
<td>2.90</td>
<td>0.29</td>
<td>5.5</td>
<td>3.5</td>
</tr>
<tr>
<td>15</td>
<td>17.1</td>
<td>5.05</td>
<td>3.39</td>
<td>0.23</td>
<td>9.9</td>
<td>2.0</td>
</tr>
<tr>
<td>20</td>
<td>25.9</td>
<td>6.40</td>
<td>4.05</td>
<td>0.20</td>
<td>14.2</td>
<td>2.2</td>
</tr>
</tbody>
</table>

processor algorithm. Results obtained for this method are shown in Table 4. In comparison with the conventional method used for finding all roots of a polynomial, the Jenkins–Traub algorithm [8], execution times achieved by the parallel quotient-difference algorithm were quite good.

6. Conclusion

A popular approach for devising parallel algorithms is the restructuring of existing serial methods of solution into a form suited to efficient execution on a parallel computer. Some techniques for performing this transformation have been described here and examples given for each. These techniques included data flow analysis, pipelining and redundancy. The execution time improvements given by the resulting parallel algorithms vary widely depending on the number of processors involved, the extent to which the parallelization has been carried out and of course the characteristics of the polynomial involved. Nevertheless the usefulness of the restructuring techniques employed has been demonstrated. Consideration should be given to making use of these techniques to try to parallelize other existing serial methods. Whether the gains offered make the use of a particular parallel algorithm worthwhile is a matter for each user to decide, based on his application and the resources available.

References

142–149.
661–674.