

Efficient Polynomial Root Finding Using SIMD Extensions

M. Moslemi, H. Ahmadi & H. Sarbazi-Azad
*IPM School of Computer Science & Sharif University of Technology,
Tehran, Iran.
azad@sharif.edu*

Abstract

In this paper, the parallel implementations of different iterative polynomial root finding methods on a processor with SIMD processing capability are reported. These methods are based on the construction of a sequence of approximations that converge to the set of roots. We have chosen four widely used methods namely Newton's, Durand-Kerner's, Aberth-Ehrlich's, and QD and implemented them using the SIMD instruction set of the Pentium processor with C++ and assembly language. Experiments show that a speedup of 3 or higher can be achieved, depending on the order of polynomial, required accuracy, and the method employed.

1. Introduction

Today, polynomial root finding is widely used in different applications. Many methods have been developed and implemented in an iterative manner. These methods construct a sequence of approximations that converge to the roots and repeat this procedure until the required accuracy is satisfied. When we are dealing with such iterative methods, one major problem is their convergence time increases with the degree of high polynomials, and usually most of those algorithms take a long time to converge. Some of such methods seem to be parallelized well, and the parallelization of them improves the convergence time to great extent.

Many efforts have been made to introduce parallel polynomial root finding methods. Like many other parallel methods [4, 9], using Single-Instruction Multiple-Data (SIMD) stream processing model is an appropriate way to implement parallel version of polynomial root finding methods. Most recent high-performance microprocessors possess SIMD processing extensions. For instance, the SIMD extension of the Intel's Pentium can perform numerous instructions over 4 single precision floating-point values packed in a single register in a parallel manner. Unfortunately, little effort has been made to exploit such capabilities in new processors. The utilization of these capabilities may

require some work on vectorizing algorithms. But it can introduce noticeable speedup in the performance of many regular computation-extensive applications, with no added cost. Here, we have vectorized 4 iterative polynomial root finding algorithms using the SIMD unit embedded within Pentium processors; the first method is Newton's method [2] which is the oldest method for extracting the roots of a given polynomial; the next one is the Durand-Kerner's method [1] which uses Weierstrass operator as an iterative function. The Weierstrass operator can only work in the case of distinct roots. The third method is Aberth-Ehrlich's method [1], [6], and the last one is the quotient difference (QD) method [2].

The rest of paper is organized as follows. Section 2 describes how different SIMD technologies allow parallel vector computation. The 4 iterative methods studied in this paper are briefly described in Section 3. In Section 4, we describe one of our implementations using SSE technology. Section 5 reports the experimental results we obtained by comparing serial and vector algorithms. Finally, conclusions and future research lines are given in Section 6.

2. The SIMD Extension of Recent Processors

The single-instruction multiple-data stream (SIMD) processing model allows the operation of a single function on multiple instances of data, instead of on a single unit of (scalar) data. Some SIMD processing extensions implemented in commercial products are Intel's MMX, streaming SIMD extension (SSE) [3], SSE2, and SSE3 technologies, AMD's 3DNow! technology, Motorola's AltiVec technology implemented in PowerPCs, and Sun's VIS technology [4].

We can assume these units as a vector processor with different vector sizes. For example 3DNow! technology uses 8 packed 64-bit register to perform SIMD operations [8]. When we use single precision floating-point data elements, 3DNow! extension appears as a 2-element vector processor, while SSE 128-bit technology can perform operation on four 32-bit floating-point values simultaneously.

The packed single-precision floating-point instructions in SSE perform SIMD operations on packed single-precision floating-point operands. Each source operand register contains four single-precision floating-point (32-bit) values, and the destination operand register contains the results of the operation performed in parallel on the corresponding values.

The scalar single-precision floating-point instructions operate on the low (least significant) double words of the two source operands. The three most significant double words of the first source operand are passed through to the destination. The scalar operations are similar to the floating-point operations in the sequential model.

3. Polynomial Root Finding Methods

In this section, four common root finding methods with different characteristics are briefly described. These methods are based on the construction of the approximation sequence. The first three methods including Newton, Durand-Kerner, and Aberth-Ehrlich need a suitable initial value to make this sequence, but the last one has a special advantage of not needing an initial value. This method is called quotient difference (QD) algorithm.

Newton's method: One of the most widely used methods of solving equations is the Newton's method [2]. This method is based on a linear approximation of the function. Consider a polynomial:

$$p(z) = \sum_{i=0}^n a_i z^{n-i}, \quad a_n \neq 0 \quad (1)$$

According to Newton's method we can approximate a new set of roots using the previous one as

$$H(z^k) = z^{k+1} \text{ and } H_i(z) = z_i - \frac{p(z_i)}{p'(z_i)} \quad (2)$$

After applying this operator for some initial root set, $Z^{(0)} = \{z_i\}$, for a number of times, we get to the roots with some error. In order to start Newton's algorithm, we should have an initial vector $Z^{(0)}$. There are some methods to determine the initial vector like Graeffe's method [2] which is useful in some special cases (when the roots are all different and widely separated).

Durand-Kerner's method: The Durand-Kerner's method is one of the simultaneous methods that allows extraction of all roots $\omega_1, \dots, \omega_n$ of a polynomial

$$p(z) = \sum_{i=0}^n a_i z^{n-i}, \text{ where } a_n \neq 0, \quad a_0 = 1, \quad a_i \in C \quad (3)$$

This method constructs a sequence, $H(z^k) = z^{k+1}$ in C^n with $Z^{(0)}$ being any initial vector and H being an

operator in C^n , constructed in such a way that $Z_i^{(k)}$ tends to root ω_i of the polynomial. A famous operator is the Weierstrass operator leading to the Durand-Kerner's method, defined as

$$H_i(z) = z_i - \frac{P(z_i)}{\prod_{j \neq i} (z_i - z_j)} \quad i = 1, \dots, n \quad (4)$$

This iteration is also called Dochav's method [7]; indeed, it is the Newton's method applied for an n -dimensional space C^n . The iterations repeat until $|Z_i^k - Z_i^{k+1}| / Z_i^k$ or $|P(z_i^k)|$ is smaller than the desired accuracy, ε . Vector initialization is important for this method because the vector elements must be distinct. A popular method to determine the initial vector is the Guggenheimer's algorithm [5] which is used to compute initial approximations before starting Durand-Kerner's algorithm.

Aberth-Ehrlich's method: The properties mentioned for the polynomial in Durand-Kerner are the same for the one in Aberth-Ehrlich's method. Again, $H(z^k) = z^{k+1}$ is the sequence leading to desired roots. The operator used with Aberth-Ehrlich's method is

$$H_i(z) = z_i - \frac{1}{\frac{P'(z_i)}{P(z_i)} - \sum_{j \neq i} \frac{1}{z_i - z_j}} \quad i = 1, \dots, n \quad (5)$$

Each component i is an iteration of the Newton's method applied to the i -th Durand-Kerner correction.

QD method: QD or *quotient difference* method is a relatively efficient method to determine all the roots of a polynomial without starting values which is its major difference with methods mentioned above. It performs the root finding with the help of two operators, e and q . At the start of the program, they are initialized to some proper initial values. For the n -th degree polynomial, we form 2 arrays of q and e terms starting by calculating the first row of q 's and e 's.

$$q_0 = -a_1 / a_0, \text{ all other } q\text{'s are zero} \quad (6)$$

$$e_i = a_{i+1} / a_i, \quad i = 1, \dots, n-1 \quad (7)$$

$$e_0 = e_n = 0 \quad (8)$$

A new row of q 's is computed as:

$$\text{New } q_i = e_{i+1} - e_i + q_i \quad (9)$$

Then a new row of e 's is computed as:

$$\text{New } e_i = \left(\frac{q_i}{q_{i-1}} \right) e_i \quad (10)$$

Repeatedly computing rows of q 's and then e 's until all e -values approach zero sufficiently the algorithm

stops. When this occurs, the q values are assumed to be the values of roots. Since this method is slow to converge, it is generally used only to get approximate values which then are improved by the Newton's method. If the polynomial has a pair of complex conjugate roots, one of the e 's will not approach zero but will fluctuate in value.

4. Parallel Implementations

Our objective is to find a parallel implementation of the described algorithms in order to compute the roots faster, especially for large polynomial degrees. The pseudo codes of all methods have been developed considering a vector processor of size of k . The implementation for a specific processor (Pentium 4), in which k equals 4, is then implemented. The less memory accesses and more vast iteration over data, make SIMD algorithms to be of higher speedup. In addition, note that pipelining may get better speedup for vector computing; this can help SSE implementation getting more advantage from processor pipeline than other a serial implementation.

Here we discuss only the parallel implementation of the Aberth-Ehrlich's method and its pseudo code; for the rest of methods a brief description of the algorithm will be given and the experimental results will be discussed in the next section.

Figure 1 presents a generic program for Aberth-Ehrlich's method. As can be seen in the pseudo code, it is presumed that n be the degree of the polynomial and k be the size of the vector processor. The main loop in the code calculating the new values of the roots in each iteration, is executed n/k times. The computation of Aberth-Ehrlich operator elements is done in the inner loops. In the first inner loop that executes n times, the values of the polynomial and its derivative at given points are computed. In the second inner loop, the sum in the divisor term of operator H for this method is calculated and finally the value of H is calculated. The main loop is repeated until a predefined accuracy (ε) is obtained.

To implement Newton's method we supposed having a polynomial of degree n and vector size of k . Then we perform the necessary computations on the vector processor to determine the roots by calculating the polynomial and its derivative value for each z_i . Consequently, the Newton operator will be carried out.

In Durand-Kerner's method, similar to the Aberth-Ehrlich's method, the polynomial value and the multiplication of root differences need to be calculated. With first simplifying inner loop, the polynomial value is determined. Moreover, the multiplication of root differences is very similar to the second inner loop.

Although QD method has different computational procedures, the concepts of vector operation apparent in the Aberth-Ehrlich's method remain the same. In the following section, we compare these methods in view of their experimental results.

```

C0...k-1 ← [1, 1, ..., 1]
T ← [ε, ε, ..., ε]
REPEAT
U0...k-1 ← [n+1, n+1, ..., n+1]
for i ← 0 to  $\frac{n}{k} - 1$ 
  D0...k-1, P0...k-1, P'0...k-1 ← [0, 0, ..., 0]
  Z0...k-1 ← [zi*k, zi*k+1, ..., zi*k+k-1]
  for j ← 0 to n-1
    A0...k-1 ← [aj, aj, ..., aj]
    P ← (P + A) * Z
    U ← U - C
    A ← A * U
    P' ← (P' + A) * Z
  A0...k-1 ← [an, an, ..., an]
  P ← P + A
  P' ← P' / Z
  for l ← 0 to n-1
    S0...k-1 ← [zl, zl, ..., zl]
    S ← Z - S
    S ←  $\frac{1}{S}$ 
    if 0 ≤ l - i * k ≤ n then Sl-i*k = 0
    D ← D + S
  H ← (Z -  $\frac{1}{\frac{P'}{P} - D}$ )
  [zi*k, ..., zi*k+k-1] ← H
UNTIL |P(H)| < T for all z;

```

Figure 1. SIMD pseudo code for Aberth-Ehrlich's method.

5. Performance Evaluation

We have implemented all of the four methods in SSE technology. For each of them we performed a comparison between sequential and parallel implementation in both best optimized C++ and assembly implementation. It is expected that the sequential assembly implementation should execute faster than that implemented in C++. According to the results this is true for some cases. However, due to certain capabilities of C++ compilers, well-optimized C++ codes result in superior performance.

In this study, two kinds of graphs can be considered: one of them based on the degree of polynomial. This speedup graphs help us to find values of n for which we will get better speedup and for which the speedup

remains almost constant, and another one illustrates the relation between speedup and the number of iterations. As we will see later in this kind of graphs, the number of iterations has no effect on the speedup and this implicitly shows good parallelization of the methods.

First, we investigate the experimental results for a constant number of iterations. We performed each algorithms 100 times and then compute the speedup. The speedup is calculated by dividing the execution time of the vector algorithm by the execution time of the equivalent serial algorithm.

The system configuration running the implemented codes is as follows:

CPU: Intel Pentium 4 Mobile Processor at 1.8 GHz
Main memory: 512 MB of DDR (266 MHz) Memory
Cache memory: 512KB L2 Cache
Operating system: Microsoft Windows XP Professional
Compiler: Microsoft Visual Studio .Net 2003

Let us start with Newton's method. As it can be observed in Figure 2, there are some ripples before speedup reaches a steady state. But for a large number of roots, speedup fluctuation reduces and a speedup of 7 is observed for the assembly and 4 for the C++ implementations. For a multiple of four, the greatest speedup is observed. This is due to the fact that we use SSE extension and as mentioned before each SSE register contains four single-precision floating point values. Hence, for a multiple of four, there is no remaining block of z_i 's in the last iteration of the $H(z)$ computation, and in these cases speedup increases to its maximum value.

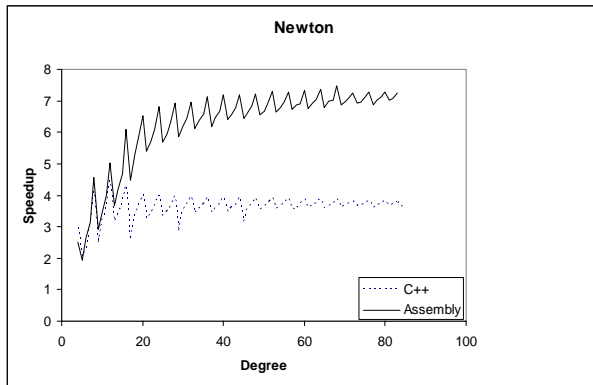


Figure 2. Speedup for Newton's method.

The next method is Durand-Kerner's; the experimental performance result is shown in Figure 3. The speedup for Durand-Kerner's method converges to 5 for assembly implementations and 4 for C++ implementations. Performing a comparison between the results presented so far, one is drawn to the conclusion that Newton's method can be parallelized better, as

greater speedup is obtained for this method. The reason of that is stem from the $\prod_{j \neq i} (z_i - z_j)$ term in the divisor of the H operator of Durand-Kerner that declines speedup.

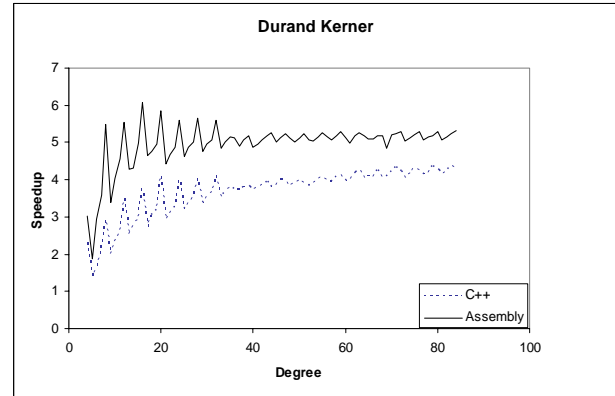


Figure 3. Speedup for Durand-Kerner's method.

The third implemented method is Aberth-Ehrlich's. Experimental results obtained from the implementation of this method are displayed in Figure 4.

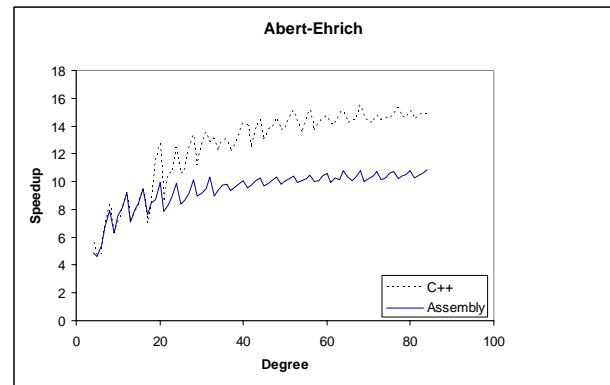


Figure 4. Speedup for Aberth-Ehrlich's method.

As shown in the pseudo code of the Aberth-Ehrlich's method, the value of $p(z_i)$ and $p'(z_i)$ are calculated in one loop and in a single memory access. Hence, the performance will be improved considerably and great speedup will be gained. The last implemented method is the QD algorithm. Results obtained from the implementation of this method are displayed in Figure 5.

The QD method has the least speedup among all of methods considered in this analysis. The reason for this is due to the operation performed to shift the register value. For example fetching a sequence starting with e_{i+1} from memory after e_i is fetched. This needs shifting and shuffling registers to put floats in the right order and then performing subtraction to calculate q . The same is repeated for the computation of e (this time for fetching

consecutive q values). These operations decrease speedup to some extent. Despite of these procedures, we get a speedup of 3.5 for both C++ and assembly implementations which would be very time saving for large values of n .

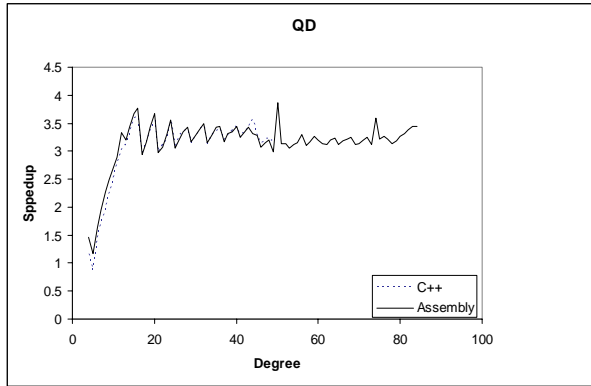


Figure 5. Speedup for QD method.

The interesting behavior of speedup curves, illustrated in Figures 2-5, shows (in some cases) a speedup of even higher than 4. It was initially predicted to be 4, as the SIMD extension we used has a 4-way organization. The reason behind this super-linear behavior for speedup is the effect of vector codes on the efficiency of processor pipelines and caches. Indeed vector codes can better exploit locality of references, pipelines, and other regular structures inside the processor.

Figure 6 shows the speedup gain for different method in question as a function of the number of iterations. Note that for all methods, the same speedup is gained by running the algorithms for different numbers of iterations. This indicates that the process of vectorizing the sequential algorithms has been effectively done.

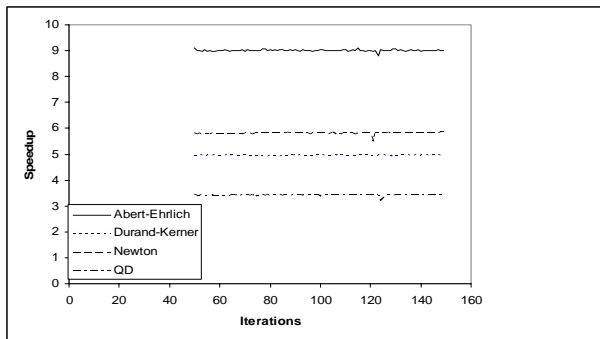


Figure 6. Speedup for different number of iterations.

6. Conclusion

Use of parallelization for polynomial root finding methods such as the Newton, Durand-kerner, Aberth-

Ehrlich, and QD algorithm has enormous effect on their execution time. We implemented this method in C++ and assembly and compared these implementations with parallel implementations in SSE extension. These implementations run 3 to 14 times for C++ and 3 to 10 for assembly faster than common sequential implementations. Thus, these implementations are completely suitable to use. Furthermore, many methods are similar to or based on the methods presented in this paper. So, the pseudo codes and algorithms can help us simply implement other polynomial root finding methods.

According to results we found out that a desirable speedup will be gained by implementing vector polynomial root finding methods because of the exposed independency in computational steps of roots. Simultaneous operations on vectors and less interaction with memory hierarchy make the algorithm to run faster in SIMD implementations. Hence, the methods use temporal operations and regular iteration over data achieves better speedup among root finding methods.

Extending our work to other types of methods used in computational mathematics and trying to parallelize them with the help of SIMD technology in order to analyze effect of vector parallelization in computational mathematics methods can be the future work in this line.

7. References

- [1] M. Cosnard and P. Fraignaud, Analysis of Asynchronous Polynomial Root Finding methods on a distributed Memory Multicomputer, *IEEE Trans. on Parallel and Distributed Systems*, Vol.5, no.6, pp.639-648, 1994.
- [2] C. F. Gerald and P.O. Wheatley, Applied Numerical Analysis, 4th ed., Addison-Wesley, New York, 1989.
- [3] Intel Corporation, Intel IA-32 Architecture Developer's Manual, 2004.
- [4] A. Strey, On the suitability of SIMD extensions for neural network simulation, *Microprocessors and Microsystems*, Volume 27, Issue 7, pp. 341-35, 2003.
- [5] R. Couturier, P. Canalda, and F. Spies, Iterative Algorithms on Heterogeneous Network Computing: Parallel Polynomial Root Extracting, *International Conference on High Performance Computing HiPC2002*, December 18-21, Bangalore, India, LNCS 2552, pp.283 - 291, 2002.
- [6] T.C.Chen and W.S.Luk, Aberth Method for the Parallel Iterative Finding Polynomial Zeros, *International Conference on APL APL94*, September 11-15, Antwerp, Belgium, p. 284, 1994.
- [7] E.D Angelova and K.I Semerdziev, Methods for the simultaneous approximate derivation of the roots of algebraic trigonometric and exponential equations, *USSR Computer Math. & Math. Phys.*, vol. 22, no. 1, pp. 226-232, 1982.
- [8] AMD Corporation, 3DNow! Developer's Manual, 2003.
- [9] S.Thakkar, T.Huff, Internet Streaming SIMD extensions, *Computer*, Vol.32, no.12, pp. 26 - 34, 1999.