Volume 4, Issue 4, April 2014 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

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

Parita Pooj, Priyanka Padmanabhan, Shweta Khushu 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].

$$F_0(x) = p_0(x),\ F_1(x) = p_0'(x) \tag{1}$$

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

$$F_2(x) = Q_1(x)F_1(x) - c_i^2 F_0(x) \tag{2}$$

$$F_{i+1}(x) = \frac{Q_i(x)F_i(x) - c_i^2 F_{i-1}(x)}{c^2_{i-1}} \tag{3}$$

This standard remainder sequence satisfies the conditions of a Sturm sequence [13]. Through the Theorem 1 [18], a pair of polynomials, say $p_1(x)$ and $p_2(x)$, 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 $(y_0, y_1, y_{2\ldots\ldots})$ interleave the roots of the original polynomial $(x_0, x_1, x_{2\ldots\ldots})$; such that intervals for calculating the roots are obtained.

$$x_0 \le y_0 \le x_1 \le y_1 \le x_2 \le y_2 \le x_3 \le y_3 \le x_{4\ldots\ldots} \tag{4}$$

For a polynomial $P_{i,j}(x)$, we define $k = \left\lfloor \frac{i+j+1}{2} \right\rfloor$ such that for $1 \le i \le j \le k \le n$, the polynomials $P_{i,k-1}(x)$ and $P_{k+1,j}(x)$ are interleaving polynomials for $P_{i,j}(x)$. 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 *Pi,j* 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.

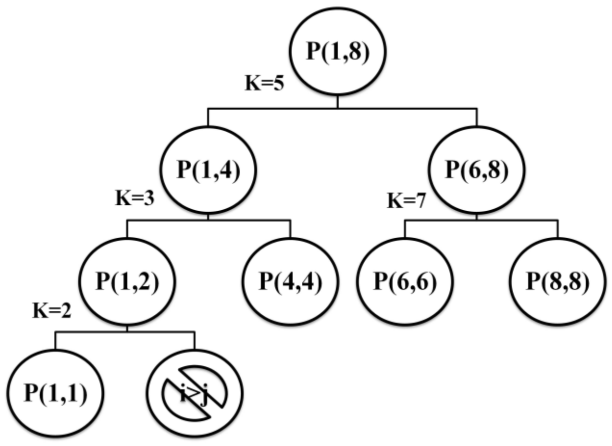

Fig.1 Polynomial tree for a polynomial with degree 8

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, $Q_1, Q_{2,\ldots}\ Q_s$ 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.

$$\begin{aligned} P_{i,j}(x) &= T_{i,j}[2,2] && 1 \le i \le j \le n \\ &= F_{i-1}(x) && 1 \le i \le n\ ;\ j = n \\ &= 1 && j \le i \end{aligned} \tag{5}$$

Where the matrices are defined as

$$S_1 = \begin{pmatrix} 0 & 1 \\ -c_1^2 & Q_1(x) \end{pmatrix} \tag{6}$$

$$S_i = \begin{pmatrix} 0 & 1 \\ \frac{-c_i^2}{c^2_{i-1}} & \frac{Q_i(x)}{c^2_{i-1}} \end{pmatrix} \quad \text{where } 2 \le i \le n\text{-}1 \tag{7}$$

$$T_{i,j} = \frac{T_{k+1,j} S_k T_{i,k-1}}{c_k^2 c^2_{k-1}} \tag{8}$$

When $k=i$, $P_{i+1,j}(x)$ 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, $y_0 = -2^m$ and $y_n = 2^m$. (9)

In every interval, the roots are calculated using certain approximation techniques. Therefore, $\tilde{y}_0, \tilde{y}_1$ .. are the approximate values and not the absolute values of the intervals. A μ-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 $2^{-\mu}$. μ-approximation to the root x_i is defined as

$$\tilde{x}_i = 2^{-\mu}\left| 2^{\mu} x_i \right| \quad (10)$$

$$\tilde{x}_i = 2^{-\mu} \times (2^{\mu} \times x_i + 0.99....)$$

$$\tilde{x}_i = x_i + 2^{-\mu} \times 0.99....$$

$$\text{For max error} \quad \tilde{x}_i \approx x_i + 2^{-\mu}$$

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 $sign(P_o(-\infty)) = (-1)^i sign(P_o(\tilde{y}_i))$

$= i + 1$ otherwise

Here, r_i denotes the number of roots of $P_0(x)$ in the interval $(-\infty, \hat{y}_i)$.

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 $P_0(x)$ at $\tilde{y}_i$ and $\tilde{y}_{i+1} - 2^{-\mu}$.
If $sign(P_o(\tilde{y}_i)) \neq (-1)^i sign(P_o(\tilde{y}_{i+1} - 2^{-\mu}))$, 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, μ bisections are performed such that the size of the interval hence obtained is $1/2^{\mu} = 2^{-\mu}$. Since μ-approximation technique was utilized, performing bisection $\lceil \log_2(b-a) + \mu \rceil$ times ensures that the obtained root is within the specified tolerance set by the value of μ. 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 μ-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 $2^{\lceil \log_2 n \rceil}$. 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 μ, 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 $P_{i,j}$ 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 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.

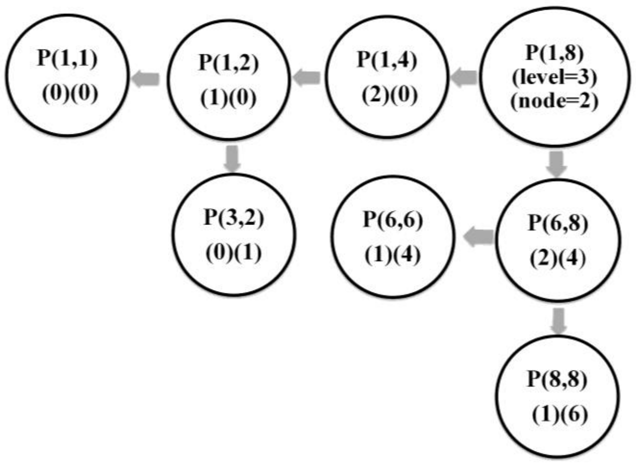

Fig 2 Level and node number associated with every stage of polynomial tree for a polynomial with degree 8

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

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 μ=8

Degree	*No of Proc.*	*Polynomial*	*Actual roots*	*Max. error of roots obtained from simulation*	*Average error of roots obtained from simulation*
2	2	$1(x^2) + 3(x) + 2$	[-2, -1]	1.14E-05	1.14E-05
3	4	$1.0(x^3) - 6.0(x^2) + 11.0(x) - 6.0$	[1, 2, 3]	0.003875673	0.00193278
4	4	$1(x^4) - 5(x^2) + 4$	[-2, -1, 1, 2]	0.003896066	0.001594537
7	8	$1.0(x^7) + 1.4(x^6) - 7.75(x^5) - 7.0(x^4) + 17.75(x^3) + 5.6(x^2) - 11.0(x) + 0.0$	[-2.5, -2, -1, 0, 1, 1.1, 2]	0.00313971	0.00122038
10	16	$1.0(x^{10}) - 4.2(x^9) - 12.95(x^8) + 73.976(x^7) - 7.112(x^6) - 294.028(x^5) + 222.51(x^4) + 273.644(x^3) - 203.448(x^2) - 49.392(x) + 0.0$	[-3.5, -2, -1, -0.2, 0, 1, 2, 2.1, 2.8, 3]	0.002081061	0.001104906
15	16	$1.0(x^{15}) - 4.5(x^{14}) - 45.24(x^{13}) + 214.47(x^{12}) + 755.9025(x^{11}) - 3910.82625(x^{10}) - 5617.76375(x^9) + 34332.0975(x^8) + 16074.645(x^7) - 149173.166(x^6) + 6681.64125(x^5) + 290320.245(x^4) - 94936.985(x^3) - 171778.32(x^2) + 77086.8(x) + 0.0$	[-4, -3.5, -3, -2.3, -2, -1, 0.5, 1, 2, 2.5, 3, 3.5, 3.8, 4]	0.003437599	0.001579975
18	32	$1.0(x^{18}) - 2.7(x^{17}) - 52.07(x^{16}) + 143.559(x^{15}) + 1060.2244(x^{14}) - 3020.52078(x^{13}) - 10682.9489(x^{12}) + 32166.535(x^{11}) + 55169.7023(x^{10}) - 184341.385(x^9) - 132231.178(x^8) + 559799.704(x^7) + 79299.0376(x^6) - 829887.076(x^5) + 165135.563(x^4) + 467009.764(x^3) - 188564.101(x^2) - 4587.66689(x) + 1869.04947$	[-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]	0.003572307	0.001328411

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 μ=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 *μ*-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 μ-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.

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

[1] B. Narendran , Prasoon Tiwari, "Polynomial root-finding: analysis and computational investigation of a parallel algorithm," Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures, p.178-187, San Diego, California, USA, June 29-July 01, 1992

[2] M. Ben-Or, E. Feig, D. Kozen, and P. Tiwari, "A Fast Parallel Algorithm for Determining All Roots of a Polynomial with Real Roots," SIAM, journal of Computing, 17(6), December 1988.

[3] C. A. Neff, "Specified Precision Polynomial Root Isolation isin NC," Proc. 91st IEEE Ann Symp. on Foundations of Computer Science, pages 152-162, 1990.

[4] M. Ben-Or and P. Tiwari, "Simple Algorithms for Approximating All Roots of a Polynomial with Real Roots," Journal of Complexity, vol. 6, 417-442, 1990.

[5] M. Cosnard and P. Fraigniaud, "Finding the Roots of a polynomial on a MIMD multicomputer," Technical report, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, February 1990.

[6] A.V. Aho, J.E. Hopcroft and J.D. Ullman, "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, MA, 1974.

[7] Jacob Philip, Faruk Kazi, and Harivittal Manglvedekar, "Quasi Steady State, Quasi Numerical Modeling of Saturated Steam-Water Spaces Using Normalized Steam and Water Properties for Efficient Computing", Fifth International Conference on Intelligent Systems, Modelling and Simulation ISMS 2014 Malaysia, pp 675-679, Jan-2014.

[8] Keny T. Lucas and Prasanta K. Jana, "Parallel algorithms for finding polynomial Roots on OTIS-torus, The Journal of Supercomputing Volume 54, Number 2, 139-153, DOI: 10.1007/s11227-009-0312-7.

[9] I. Foster, "Designing and Building Parallel Programs," Addison-Wesley, 1995.

[10] D. J. Tylavsky and A. Boss, et al., "Parallel processing in power systems computation", IEEE Trans. on Power Systems, vol. 7, no. 2, pp.629 -637 1992.

[11] G.E. Alefeld, F.A. Potra, and Y. Shi, "Algorithm 748: Enclosing zeros of continuous functions," ACM Transaction on Mathematical Software, vol. 21, no. 3, pages 327–344, September 1995.

[12] K Geroge H. Ellis and Layne T. Watson, "A parallel algorithm for simple roots of polynomials," Comps. & Maths with Appls., Vol. 10, No. 2, pp. 107-121, 1984.

[13] N. Jacobson, "Basic Algebra I," Freeman, San Francisco, 1974.

[14] George E. Collins, "Subresultants and Reduced Polynomial Remainder Sequences," Journal of the ACM (JACM), vol. 14, no. 1, pp. 128–142, Jan. 196.

[15] A. Edelman and H. Murakami, "Polynomial roots from companion matrix eigen values," *Math. Comput.*, vol. 64, no. 210, pp.763 -776, 1995.

[16] MPI Forum.MPI: A message-passing interface standard, version 2.2, September 4th 2009. www.mpi-forum.org.

[17] Elts Ekaterina, "Comparative analysis of PVM and MPI for the development of physical applications on parallel clusters," Saint-Petersburg State University, 2004.

[18] B. Narendran, P. Tiwari, "Polynomial root-finding: analysis and computational investigation of a parallel algorithm," Computer Sciences Technical Report no. 1061, University of Wisconsin Madison, Wisconsin, Dec. 1991.

[19] Indiana University, "Open MPI: Open Source High Performance Computing,"Internet: https://www.openmpi.org/doc/v1.4/man1/mpirun.1.php, Feb. 14 2012 [March 10 2014]

[20] Indiana University, "Open MPI: Open Source High Performance Computing," Internet: https://www.open-mpi.org/doc/v1.4/man1/mpiexec.1.php , Feb. 14 2012 [March 10 2014]

[21] Timothy J. Rolfe, "A Specimen of Parallel Programming: Parallel Merge Sort Implementation," ACM Inroads, Vol. 1, Issue 4, pages 72-79, December 2010.

[22] Lisandro Dalsin, "MPI for Python," Internet: http://mpi4py.scipy.org/ , [March 10 2014]