Implementing the Parallel Quasi-Laguerre's Algorithm for Symmetric Tridiagonal Eigenproblems

T. Y. Li[1]* and Xiulin Zou[2]

[1] Department of Mathematics, Michigan State University, E. Lansing, MI48824, USA
[2] Department of Math. Sciences, Oakland University, Rochester, MI 48309, USA

Abstract. In this article, a fully scalable parallel algorithm is presented for solving symmetric tridiagonal eigenvalue problems using quasi-Laguerre's method. The algorithm is implemented using PVM and tested on a variety of matrices with a load balancing scheme. Test results show that the algorithm has high parallel efficiency. Compared with other existing algorithms, our algorithm seems to be the best for distributed memory parallel architecture.

1 Introduction

Let T be a symmetric tridiagonal matrix of the form

$$T = [\beta_{i-1}, \alpha_i, \beta_i] \tag{1}$$

where $\beta_i \neq 0$ for $i = 1, \cdots, n-1$. The eigenvalues of T are the roots of the characteristic polynomial

$$f(\lambda) = det(T - \lambda I). \tag{2}$$

Laguerre's iteration for finding all the eigenvalues of T was first studied in [7], demonstrating clear advantages over other existing algorithms for the same problem. A parallel version of the algorithm was reported in [6]. Later, quasi-Laguerre's algorithm was established in [3] and gained more speedup over Laguerre's iteration. In this paper, we shall present the parallel version of the quasi-Laguerre's iteration for solving all roots of $f(\lambda)$ in (2). The algorithm, including a load balancing scheme, is implemented using PVM(Parallel Virtual Machine [4]) on a cluster of workstations. Numerical results on a substantial variety of matrices show that our algorithm is faster than the bisection/multi-section method(DSTEBZ in LAPACK [1]) under any circumstances and is faster than the root free QR (DSTERF in LAPACK) when four or more processors are available.

* The research was supported in part by NSF under Grant DMS-9504953 and a Guggenheim Fellowship.

2 The quasi-Laguerre's iteration formula

Let $f(x)$ be a polynomial with only real roots. Assume x_0 and x_1 are two real points with no root of $f(x)$ between them. Let $q_i = f'(x_i)/f(x_i)$, for $i = 0, 1$. Then the quasi-Laguerre's iteration formula [9] with multiplicity index m is given by

$$QL_{\pm} = \frac{x_0 + x_1}{2} + \frac{mn - [S + m\frac{\Delta q}{\Delta x}]\frac{(\Delta x)^2}{4}}{-m\frac{(q_0+q_1)}{2} \pm \sqrt{-m(n-m)S + S^2\frac{(\Delta x)^2}{4}}}, \tag{3}$$

where $\Delta q = q_1 - q_0$, $\Delta x = x_1 - x_0$ and $S = q_0 q_1 + n\frac{\Delta q}{\Delta x}$.

The quasi-Laguerre's iterative method converges monotonically and globally with super-linear convergence rate if the polynomial has only real roots and the multiplicity index matches the multiplicity of the root [2] [9]. A multiplicity estimation formula is introduced in [9] to approximate the multiplicity of the nearest root and to accelerate the convergence of the iteration.

3 The Split Merge algorithm

Let $\lambda_1 < \lambda_2 < \cdots < \lambda_n$ be the zeros of f in (2). To use our quasi-Laguerre's iteration to approximate any λ_i, $i = 1, 2, \cdots, n$, it is essential to find a pair of starting points $x_{(0)}$ and $x_{(1)}$, with no λ_j's lying between them. For this purpose, we *split* T into

$$\hat{T} = \begin{pmatrix} T_0 & 0 \\ 0 & T_1 \end{pmatrix} \tag{4}$$

where

$$T_0 = \begin{pmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \ddots & \ddots & \\ & \ddots & \ddots & \beta_{k-1} \\ & & \beta_{k-1} & \alpha_k - \beta_k \end{pmatrix}, \quad T_1 = \begin{pmatrix} \alpha_{k+1} - \beta_k & \beta_{k+1} & & \\ \beta_{k+1} & \ddots & \ddots & \\ & \ddots & \ddots & \beta_{n-1} \\ & & \beta_{n-1} & \alpha_n \end{pmatrix}. \tag{5}$$

Obviously, the eigenvalues of $\hat{T}$ consist of eigenvalues of T_0 and T_1. Without loss of generality, we may assume $\beta_i > 0$, for all $i = 1, 2, \cdots, n-1$ [7]. The following interlacing property [5, Theorem 8.6.2, p462] for this rank-one tearing is important to our algorithm.

Theorem 1. *Let $\lambda_1 < \lambda_2 < \cdots < \lambda_n$ and $\hat{\lambda}_1 \le \hat{\lambda}_2 \le \cdots \le \hat{\lambda}_n$ be eigenvalues of T and $\hat{T}$ respectively. Then*

$$\hat{\lambda}_1 \le \lambda_1 \le \hat{\lambda}_2 \le \lambda_2 \le \cdots \le \hat{\lambda}_n \le \lambda_n < \hat{\lambda}_{n+1} \tag{6}$$

with the convention $\hat{\lambda}_{n+1} = \hat{\lambda}_n + 2\beta_k$.

The eigenvalues of $\hat{T}$ in (4) consist of eigenvalues of T_0 and T_1 in (5). To find eigenvalues of T_0 and T_1, the split process described above may be applied again. Indeed, the splitting process can be applied to T recursively until 2×2 and 1×1 matrices are reached.

To *merge*, we use eigenvalues of $\hat{T}$ to approximate eigenvalues of T by using quasi-Laguerre's iteration. Interlacing in (6) provides good information for picking two valid initial points for the quasi-Laguerre's iteration.

4 The parallel quasi-Laguerre's method

We implemented the parallel quasi-Laguerre's algorithm using PVM with a master and slave program. The master program divides the spectrum into small chunks, $1 : n\_1$, $(n\_1 + 1) : n\_2$, $\cdots$, $(n\_k + 1) : n$, where $i : j$ denotes the eigenvalues from number i to number j. Then the master spawns the slave process to all available machines that form the PVM machine, and sends out information to the slaves. The information sent to the slaves includes the matrix order, the diagonal and off diagonal entries of the symmetric tridiagonal matrix, starting numbers and ending numbers of eigenvalue chunks, and some other administrative information such as parent process ID and slave process IDs and so on. The master program also serves as an administrator that is excluded from computation and also processes other jobs that has to be done sequentially. So we don't spawn slave process onto the machine on which the master process is running.

The slave program spawned by the master program receives data and 'instructions' sent by the master program and calls split-merge Quasi-Laguerre's subroutine to find the respective eigenvalues of the matrix, then sends the results back to the master program, and wait for another chunk of eigenvalues to compute until an exit instruction is received.

For a slave process to compute the eigenvalue chunk $i : j$, it follows the following steps.

(1). Compute the Gershgorin circle(interval) $[lb, ub]$ that contains all the eigenvalue of T.

(2). Use bisection on interval $[lb, ub]$ and Sturm sequence to find a best interval $[a_i, b_j]$ that contains the i^{th} to j^{th} eigenvalue(s) of T. We consider an interval 'best' in the following sense: for well separated eigenvalues, $[a_i, b_j]$ contains the i^{th} to j^{th} eigenvalue of T only; in case of a cluster, $[a_i, b_j]$ is the smallest interval within the error tolerance that contains the i^{th} to j^{th} eigenvalue of T.

(3). Split the matrix level by level, as described in section 3, until one by one or two by two matrices are obtained.

(4). Starting from the bottom level of the tree, find the eigenvalue(s) in $[a_i, b_j]$ of each 1×1 or 2×2 matrix. Then use the quasi-Laguerre's method to compute the eigenvalue(s) in $[a_i, b_j]$ of the matrices in the second level (from bottom up), and then the third level, and so on until the top level that is the matrix T.

During this procedure, no processor needs to communicate with other processors to exchange information, therefore it constitutes a fully parallel and scalable algorithm.

5 Testing matrices

We used the following matrices to test our parallel code.

Type 1. Toeplitz matrices $[b, a, b]$.
Type 2. $\alpha_1 = a - b$, $\alpha_i = a$ for $2 \leq i \leq n-1$, $\alpha_n = a + b$. $\beta_j = b$, $1 \leq j \leq n-1$.
Type 3. $\alpha_i = a$ for i odd, $\alpha_i = b$ for i even. $\beta_i = 1$.
Type 4. $\alpha_i = 0$, $\beta_i = \sqrt{i(n-i)}$.
Type 5. $\alpha_i = -[(2i-1)(n-1) - 2(i-1)^2]$, $\beta_i = i(n-i)$.
Type 6. Wilkinson matrices W_n^+. (see [8])
Type 7. Random matrices. α_i's and β_i's are generated by a random number generator.

6 Load balancing

For a PVM machine that is composed of a cluster of general purpose shared workstations interconnected by Ethernet cable or FDDI cable, load balancing seems inevitable.

Here are two experimental results that exhibit the uneven computation time among the processors due to the shared environment and heterogeneous environment. The first experiment was done on six DEC Alpha workstations(one master and five slaves) in a shared environment, that is, all the processes that compute the eigenvalues of a matrix must share CPU with other CPU-intensive processes (mostly from other users). The second experiment was done in a heterogeneous environment, six DEC Alpha-s and two SUN Sparc10s(one Alpha machine served as the master and all other 7 machines served as slaves). The CPU clock speeds for the DEC Alpha workstations (model 3000/400) and SPARC10 workstations used for the experiment are 133MHz and 33MHz, respectively.

In a shared environment, it is somewhat difficult to reproduce an experiment since other user's processes come and go randomly. To make the comparison more meaningful, both experiments were conducted during a reserved time period while no other user can get onto the system. In the first experiment, a shared environment is created by creating two CPU-intensive jobs(called dummy processes) on one of the slave machines, then run the parallel quasi-Laguerre program on the PVM machine. One of the slaves must share CPU with the other two dummy processes that are running on the same host, hence it gets only 1/3 of the CPU access. Both experiments compute all the eigenvalues of a 5000 by 5000 type 4 matrix. The time of each slave without load balancing is plotted in Figure 1 and Figure 2.

The following ideas are implemented in designing the load balancing scheme for our parallel quasi-Laguerre method.

- Create uneven loads so that earlier distributed load has slightly large chunk size than later distributed loads. This method has an effect of balancing the job as a whole. Also the process which gets the last job won't take too long to finish since the last job is the smallest in chunk size.

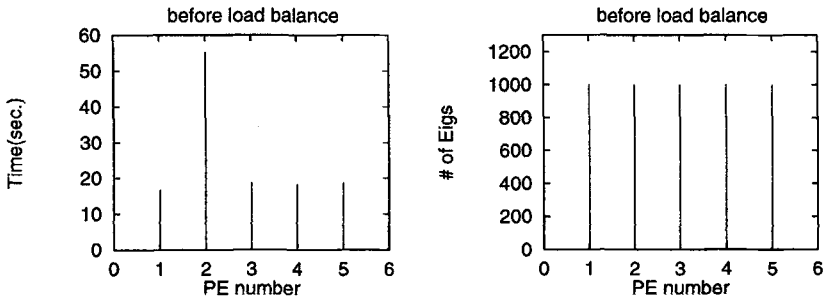


Fig. 1. Before load balancing in shared environment, other CPU intensive jobs are running on PE # 2 also. Left: Execution time of each Alpha workstation on type 4 matrix of size 5000. Right: Number of eigenvalues computed by each Alpha workstation.

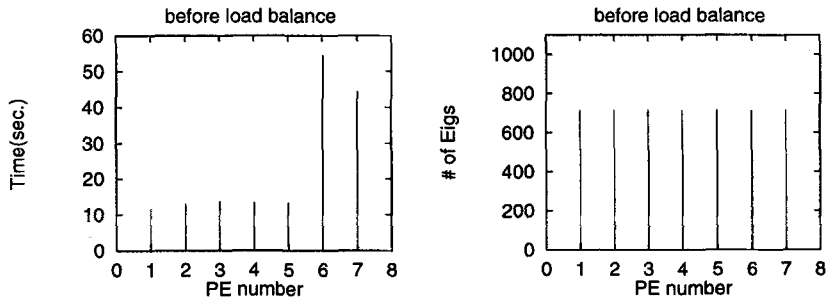


Fig. 2. Before load balancing in Heterogeneous environment, PE ## 1-5 are Alpha workstations, PE ## 6-7 are SUN Sparc10s. Left: Execution time of each workstation on type 4 matrix of size 5000. Right: Number of eigenvalues computed by each workstation

- Create more subtasks than the number of available host machines so that faster processes can finish more jobs.
- Spawn more process to each host to gain more CPU favor.
- Reset process priority level to a lower level for courtesy of the actual workstation owners. Lower process priority number means less CPU access, hence less intrusion to the actual workstation owners.

Our algorithm incorporated all of the above features, and the user can control the situation by choosing appropriate parameter values to run the program. We only discuss the first two items here, create more and uneven loads, since the other two items are more situation dependent. We used two parameters, *n_rounds*(number of rounds to distribute the subtasks) and *diff_size*(chunk size difference between successive processes), to determine how many rounds (each round has *n_hosts* subtasks) of subtasks to create and how much difference in chunk size between the successive chunks. First of all, the total number of eigenvalues is split into nearly equal chunks(with difference of at most one), then use the value *n_rounds* to further divide the chunks into smaller ones and use the value of *diff_size* to

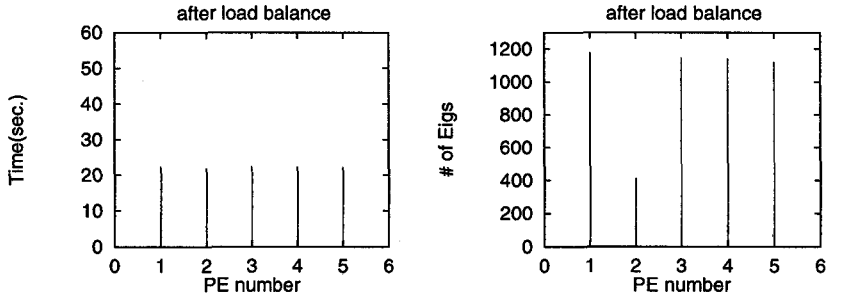


Fig. 3. After load balancing in shared environment, other jobs are running on PE # 2 also. Left: Execution time of each workstation on type 4 matrix of size 5000. Right: Number of eigenvalues computed by each Alpha workstation.

create difference among the chunk sizes. In this way, a job queue is established with chunk sizes in descending order. The rest of the program distributes the chunks from this queue to the slave processes until the queue is empty.

Since the whole job is divided into many small chunks to create more and smaller subtasks, each job takes less time to finish and the processors that finish earlier can get more subtasks to process. As a whole, every host contributes and the hosts that have less load(from other users) or faster CPU speed contribute more. Hence, an overall balanced timing distribution is achieved. The experiment results with this load balancing scheme is plotted in Figures 3 and 4. Notice that time(in seconds) spent on computing is balanced among the participating processes while the number of eigenvalues computed by each host becomes different.

Experiments showed that more subtasks create more overheads. In a homogeneous environment, uniform subdivision works slightly better than the nonuniform subdivision method. But in a heterogeneous environment or a shared platform, this load balancing scheme demonstrates a great advantage.

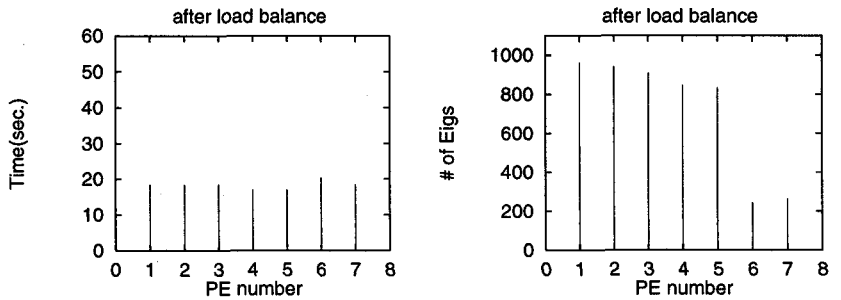
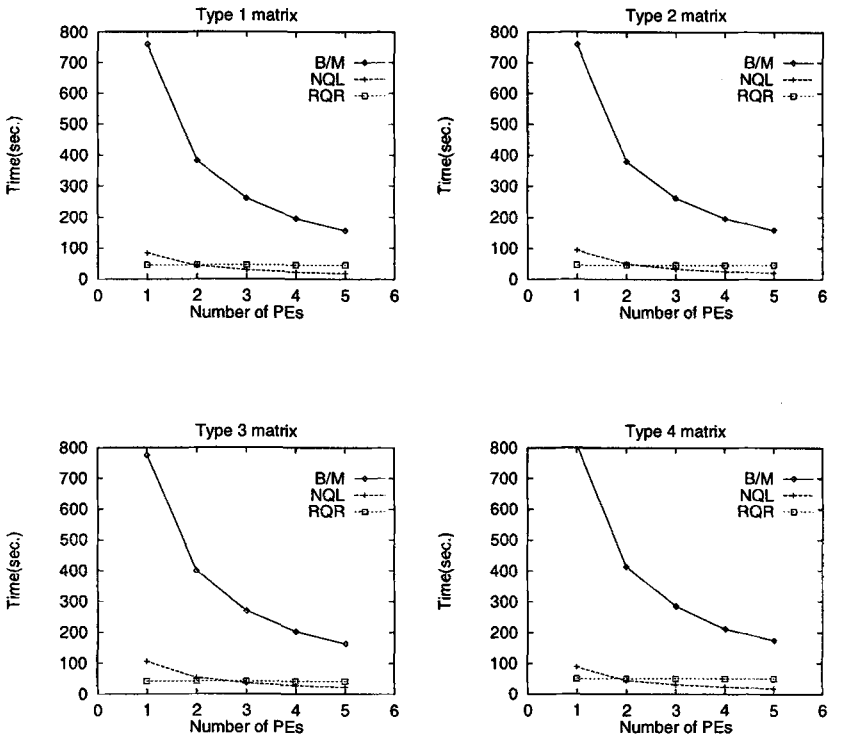


Fig. 4. After load balancing in Heterogeneous environment, PE ## 1-5 are Alpha workstations, PE ## 6-7 are SUN Sparc10s. Left: Execution time of each workstation on type 4 matrix of size 5000. Right: Number of eigenvalues computed by each workstation.

7 Comparison with parallel bisection and sequential root free QR

We tested type 1 to 7 matrices of order 5000 on six DEC Alpha workstations. We run the parallel program(quasi-Laguerre and bisection) on different number of host machines to compute all the eigenvalues of the seven types of matrices of order 5000. We also run the root free QR program from LAPACK. The total time for each run is recorded and the results are plotted in Figure 5. Root free QR method could not take advantage of all available machines. For matrices of types 1-5, our parallel quasi-Laguerre's algorithm beats root free QR with three or more machines, whereas for type 6 matrix and type 7 matrix we need four and five machines respectively to lead in speed. In all cases, the quasi-Laguerre's iteration outperforms bisection method.



References

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorenson, *LAPACK User's Guide*, SIAM, Philadelphia, 1992.

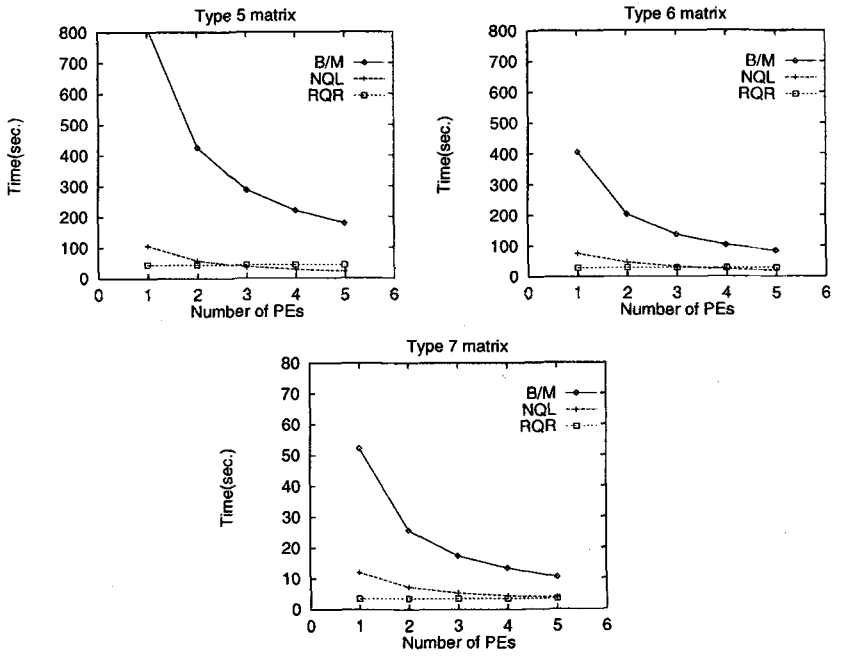


Fig. 5. Comparison between bisection – B/M, quasi-Laguerre – NQL, and root free QR – RFQR, on seven types of matrices of order 5000

2. Q. Du, M. Jin, T. Y. Li and Z. Zeng, *The Quasi-Laguerre iteration*, to appear: Math. Comp.
3. Q. Du, M. Jin, T.Y. Li and Z. Zeng *Quasi-Laguerre iteration in solving symmetric tridiagonal eigenvalue problems*, to appear: SIAM J. Sci. Comput.
4. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM 3 User's Guide and Reference Manual*, September, 1994.
5. G. H. Golub and C. F. Van Loan, *Matrix Computations, 2nd Ed.*, The Johns Hopkins University Press, Baltimore, MD, 1989.
6. C. Trefftz, P. McKinley, T. Y. Li, and Z. Zeng, *A scalable eigenvalue solver for symmetric tridiagonal matrices*, Parallel Computing, 21(1995), pp. 1213-1240.
7. T. Y. Li and Z. Zeng, *Laguerre's iteration in solving the symmetric tridiagonal eigenproblem — revisited*, SIAM J. Sci. Comput., Vol. 15, No. 5 (1994), pp. 1145-1173.
8. J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford University Press, Oxford, 1965.
9. X. Zou, *Quasi-Laguerre's method and its parallel implementation on solving symmetric tridiagonal eigenvalue problems*, Ph.D thesis, Michigan State University, 1995.