Design and Performance Evaluation of a Distributed Eigenvalue Solver on a Workstation Cluster*

C. Trefftz, C.C. Huang, P.K. McKinley
Department of Computer Science
Michigan State University
East Lansing, Michigan 48824

T.Y. Li, Z. Zeng
Department of Mathematics
Michigan State University
East Lansing, Michigan 48824

Abstract

Clusters of high-performance workstations are emerging as promising platforms for parallel scientific computing. This paper describes an eigenvalue solver for symmetric tridiagonal matrices, as implemented on a cluster of workstations using two different interprocess communication packages, PVM and P4. The algorithm is based on the *split-merge* technique, which uses Laguerre's iteration and exploits the separation property of rank two splitting in order to create subtasks that can be solved independently. A performance study that compares the distributed, parallel split-merge algorithm to a parallel version of the well-known bisection algorithm, over standard matrix types, demonstrates the performance advantage of the new algorithm and its cluster implementation.

1 Introduction

As quantitative analysis becomes increasingly important in the sciences and engineering, the need grows for faster and more efficient methods to solve eigenvalue problems. Large eigenvalue problems occur in a wide variety of applications, including the dynamic analysis of large-scale structures such as aircraft and ships, prediction of structural responses in solid and soil mechanics, the study of solar convection, modal analysis of electronic circuits, and the statistical analysis of data. Solving for the eigenvalues of large systems is a computationally-intensive task that may need to be carried out many times within a particular application; reducing its execution time will improve the performance of the application.

Solving large numerical applications has traditionally been one of the tasks best suited for supercomputers. Recently, parallel processing, which can greatly decrease the time required to solve many problems, has started to dominate the supercomputer industry. One trend in supercomputer architectures has been towards the use of *massively parallel computers* (MPCs).

As an alternative to MPCs, there has recently emerged increasing interest in using clusters of workstations for parallel scientific computing. The reasons for the popularity of these so-called "distributed supercomputers" are several [1]. First, clusters are often more economical than either a traditional vector-based supercomputers or MPCs. Second, the memory capacity of each workstation is typically much greater than that of an MPC node, allowing large problem instances to be addressed. Third, the I/O capacity of the system is larger than that of an MPC because each workstation has its own disks. Fourth, a cluster is more flexible than an MPC: additional computing and communication capacity, in the form of new workstation models and faster networks, can be easily configured into the system. Finally, clusters can be used for parallel and distributed computing at the same time that they are used as workstations to meet the computing needs of individuals. Such sharing of resources (both processors and communication links) can also be considered a drawback of clusters, however, since it is likely to reduce the performance of a particular application. Another disadvantage of clusters is that communication latency is often much longer than in an MPC, particularly if conventional networking technology is employed. Distributed parallel applications must accommodate these characteristics.

In this paper, we report the results of designing a distributed, parallel eigenvalue solver to execute efficiently on a cluster of workstations. Our current efforts focus on the special case of finding the eigenvalues for symmetric tridiagonal matrices. As one of the most fundamental problems of computational mathematics, this problem continues to receive considerable attention in the literature due to its wide applicability. Several parallel and distributed algorithms have been

*This work was supported in part by DOE grant DE-FG02-93ER25167, by NSF grants MIP-9204066, CCR-9024840, CDA-9121641, and CDA-9222901, and by an Ameritech Faculty Fellowship.

1063-6927/94 $03.00 © 1994 IEEE

developed to address this problem [2, 3, 4, 5, 6]. We study the ***split-merge algorithm***, designed originally for shared-memory parallel architectures by Li and Zeng [7]. This algorithm is inherently parallel and takes advantage of a fast iteration technique, namely, Laguerre's method.

We have previously designed and implemented a version of this algorithm for MPCs [8]. Using dynamic load balancing and efficient communication routines, we demonstrated good speedup of the algorithm when implemented on a 64-node nCUBE-2 hypercube. In the project described herein, the split-merge algorithm was redesigned and implemented atop a cluster of workstations interconnected through a simple Ethernet LAN. The purpose of the project is twofold: First, faster methods are sought by which to solve eigenvalue problems. Second, efficient solutions are sought to the communication and scheduling problems that arise when executing numerical scientific applications in distributed environments.

The remainder of the paper is organized as follows. In Section 2, a brief overview is given of the mathematical foundations for this work, and the sequential split-merge algorithm is described. Section 3 describes our basic approach to parallelizing the split-merge algorithm in the earlier MPC study. In Section 4, the cluster-based implementations of the algorithm are described and compared with the MPC version. Evaluations of several aspects of the distributed implementation are presented, with emphasis on the effects of different load balancing strategies, communication overhead, and interference from other user processes. Section 5 presents the results of a performance evaluation study of the distributed implementations, including execution times on well-known matrices and comparisons with a distributed implementation of the bisection algorithm [6]. Section 6 concludes the paper.

2 Foundations

The problem of finding the eigenvalues of a matrix can be stated as follows: Find the values λ that satisfy the equation: $A\mathbf{x} = \lambda\mathbf{x}$ for a vector $\mathbf{x}$, which is called an eigenvector. The values λ are eigenvalues. The problem can be rewritten as follows: Given a matrix A, solve $f(x) = det[A - \lambda I] = 0$. Symmetric tridiagonal (ST) matrices have the form shown in Figure 1(a). All nonzero entries occur on either the main diagonal, the superdiagonal, or the subdiagonal. Furthermore, the superdiagonal is identical to the subdiagonal.

Finding the eigenvalues of symmetric matrices is a common problem found in many different fields. A procedure commonly used to solve this problem is to reduce the original "full" matrix to a tridiagonal matrix. This operation is achieved by premultiplying the matrix by an orthogonal matrix U and postmultiplying it by U^T. The matrix U is chosen so that it introduces 0's in the original matrix except in the main diagonal, the superdiagonal and the subdiagonal. The eigenvalues of the original matrix and the reduced matrix are identical. Thus, finding the eigenvalues of a symmetric tridiagonal matrix is crucial in the process of finding the eigenvalues of symmetric matrices.

$$\begin{bmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \beta_3 & \alpha_3 & & \\ & & & \cdots & \beta_n \\ & & & \beta_n & \alpha_n \end{bmatrix} \qquad \left[\begin{array}{c|c} A_1 & 0 \\ \hline 0 & A_2 \end{array}\right]$$

(a) ST matrix A (b) ST matrix A'

Figure 1. Matrices in split-merge eigenvalue solver

The split-merge algorithm relies on the so-called ***separation property*** [9] in order to find the eigenvalues of ST matrices. Given an ST matrix A with nonzero subdiagonal elements, this technique uses the matrix A' produced by replacing some β_i with 0, as shown in Figure 1(b). Let $\lambda_1 < \lambda_2 < ... < \lambda_n$ be the eigenvalues of A, which are to be found. Assume that it is possible to find the eigenvalues of the submatrices A_1 and A_2. Let $\lambda_1^0 \leq \lambda_2^0 \leq ... \leq \lambda_n^0$ be the eigenvalues of A_1 and A_2 after they have been found, merged, and sorted in increasing order. The separation property states that $\lambda_{i-1}^0 < \lambda_i < \lambda_{i+1}^0$ and that $\lambda_{i-1} < \lambda_i^0 < \lambda_{i+1}$ for all values of i.

In the split-merge algorithm, the separation property is used as follows: the eigenvalues of matrices A_1 and A_2 are found and then used as the initial approximations to the eigenvalues of matrix A. The process may be applied recursively in a divide-and-conquer manner until matrices of sizes 2×2 are reached, for which eigenvalues may be found easily. Without loss of generality, assume that n, the order of the matrix, is a power of 2; specifically, let $n = 2^k$. The algorithm proceeds in a series of k stages. In the first stage, the eigenvalues for each of the 2×2 arrays are found using the quadratic formula. Results from pairs of neighboring subarrays are merged, sorted, and used in solving 4×4 arrays in the second stage. The same procedure is repeated for the following stages: At stage i, eigenvalues are found for 2^{k-i} submatrices of size 2^i, using the results of stage $i-1$ as initial approximations.

In order to find an individual eigenvalue from an initial approximation, the split-merge algorithm uses Laguerre's method for finding the zeroes of a polynomial with real and simple zeros. This method exhibits a cubic convergence rate. The split-merge algorithm, which is described in detail in [7], operates as shown in Figure 2.

Algorithm 1: Split-Merge
Input: Symmetric tridiagonal matrix A of order 2^k
Output: The eigenvalues λ_i, $i = 1, \ldots, 2^k$, of A
Procedure:
begin
  Find eigenvalues for the 2^{k-1} submatrices of A of size 2×2.
  for $i = 2$ **to** k
    for $j = 1$ to 2^{k-i}
      Combine submatrices $2j - 1$ and $2j$ of order 2^{i-1} into one matrix of size 2^i. (That is, merge their eigenvalues to form a list of initial approximations for Laguerre's iteration method in the next stage.)
      for $\ell = 1$ to 2^i
        Use Laguerre's iteration to find λ_ℓ for the jth submatrix of order 2^i.
      endfor
    endfor
  endfor
end

Figure 2. Split-merge algorithm.

The algorithm can be viewed as a "tree" of tasks, as shown in Figure 3. At the leaves of the tree, the eigenvalues for the 2×2 submatrices are found. These results are merge-sorted in a pairwise fashion, and in the next stage, the eigenvalues for the 4×4 submatrices are found. This process continues until the eigenvalues for the original matrix are produced at the root of the tree.

3 Distributing the Workload

Although the split-merge algorithm can be viewed logically as a tree, the nodes in this tree clearly should not also represent processors in a parallel and distributed implementation. If this were the case, for example, a single node would be solely responsible for executing the entire last stage of the algorithm, severely reducing the parallelism of the algorithm. In parallelizing the split-merge algorithm, a key objective is to maximize the fraction of time that each processor

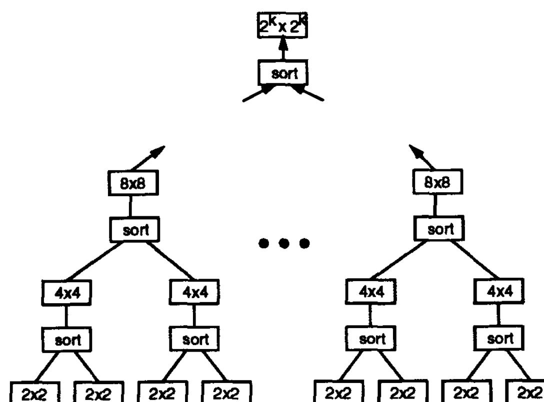


Figure 3. Representation of the split-merge algorithm

is busy solving for eigenvalues in each of the k stages. For purposes of discussion, let us assume that the number of processors and the order of the input matrix are both a power of two, 2^d and 2^k, respectively. The method also works if these conditions do not hold.

3.1 Original Implementation

In our initial approach, the responsibility for solving for eigenvalues was divided evenly among processors, that is, at all stages, every processor is responsible for finding 2^{k-d} eigenvalues. In the first $k - d$ stages, no communication is necessary between processors, since each submatrix is small enough to be handled by a single processor. Specifically, each node i initially performs the first $k - d$ stages independently, thereby solving for the 2^{k-d} eigenvalues of the ith (numbering from left to right) submatrix of order $k - d$. In the last d stages, nodes must communicate their results to other nodes where those results are merged and sorted; the appropriate sets of eigenvalues are returned to each node in order to be used as input to the next stage. The processing elements (nodes) are divided in two categories: sinks and clients. Henceforth, the first $k-d$ stages of the algorithm will be referred to as the *local* stages, and the last d stages will be referred to as the *distributed* stages.

This approach, which was originally implemented on a 64-node nCUBE-2 hypercube multicomputer, did not perform particularly well. Investigation revealed that the time required for processors to finish their share of eigenvalues was affected by the relatively high variance in the number of iterations needed to find eigenvalues. Although the majority of the eigenvalues require a single iteration, a significant number require more iterations. In the distributed stages of the

algorithm, where communication between processors is necessary, this imbalance caused those processors that completed their tasks early to remain idle, waiting for others to finish. This unexpected phenomenon greatly reduced the efficiency of the algorithm, thereby limiting speedup.

3.2 Use of Dynamic Load Balancing

The previous result led to a redesign of the algorithm in an attempt to achieve better load balancing among processors. The load balancing algorithm is centralized and receiver-initiated [10]. The general strategy works as follows. Within a given stage, each node is initially required to solve only a fraction of the eigenvalues that it would have been responsible for in the original algorithm. One node (node 0) serves as the *coordinator* and is responsible for managing allocation of the remaining eigenvalues to nodes that finish early. When a node completes its initial workload, it sends its results to the coordinator, which may dispense an additional set of initial eigenvalue approximations to that node, called a ***subsequent workload.*** This process repeats until all the eigenvalues have been found for the current stage. Node 0 collects the results from the other nodes and then sends the respective results back to the sinks, which are responsible for merging and sorting the eigenvalues solved at each stage.

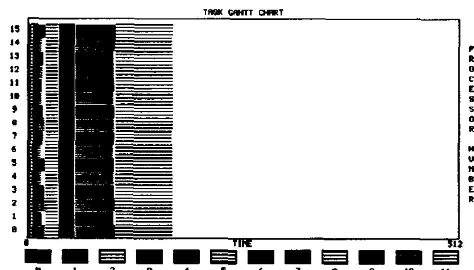


Figure 4. Trace of load balancing algorithm

This approach was effective in reducing the load imbalance among the nodes cooperating within each stage. Figure 4 shows a trace of the execution of the modified algorithm on a 2048 × 2048 matrix. Shaded areas indicate when processors are busy. In this example, the algorithm requires 11 stages. Stages 1 though 7 execute without interprocessor communication and are represented at the far left of the figure. Stages 8 through 11 require cooperation (and hence communication) among nodes. The nodes finish each stage at approximately the same time, resulting in good efficiency. The speedup of the algorithm is sensitive to the sizes of the initial and subsequent workloads, however. If the workloads are too small, then too much time is spent in communication between the clients and the coordinator. In addition, the coordinator may not be able to service all the outstanding messages immediately, delaying those nodes requesting additional work. On the other hand, if the workloads are too large, then the phenomenon of load imbalance appears again.

4 Cluster Implementation

In order to study the split-merge algorithm in a distributed environment, it was implemented and tested on a cluster of Sun Sparc-10, model 30 and model 40 workstations, interconnected by a typical Ethernet network. The workstations used for the experiments were also available for general use by students and faculty, that is, other users could freely access the workstations at any time. This environment is consistent with our goal of testing the hypothesis that general-purpose, shared workstations can provide competitive performance for scientific computing tasks.

The eigenvalue algorithm was implemented using two different programming environments, PVM and P4. PVM [11], a public domain package from Oak Ridge National Laboratory, provides a software infrastructure for network-based heterogeneous concurrent computing. P4 [12], developed at Argonne National Laboratory, comprises a library of macros and subroutines that support monitors for shared-memory programming, message-passing primitives, and support for heterogeneous cluster computing. Since there were no significant differences in performance between the PVM and P4 implementations, only the PVM implementation is described here.

4.1 Environmental Differences

Porting the nCUBE-2 code to PVM (version 2.4, and later version 3.1) on the cluster was relatively straightforward. Both environments are based on message passing, and their functionality is similar. Both environments support programming in C and Fortran and allow the exchange of messages between program components written in either language. However, some redesign of the algorithm was necessary in order to accommodate three major differences between the two environments: allocation of system resources to applications, communication latency, and processor speed.

In the nCUBE-2, full subcubes are allocated to each application. One instance of the program runs on each node of the subcube, and the hypercube topology and the underlying routing algorithm prevent com-

munication conflicts among the messages of different programs. In PVM, the processes constituting the distributed application are allocated to the available workstations. The number of process instances can be larger than the number of workstations available; the benefits of this strategy will be described later. Also, executing parallel applications on a cluster of general-use workstations introduces randomness in the performance of the program from two sources: the load of the workstations and the load of the network.

The communication latencies of the two environments are very different. According to traces of program executions, the round-trip message latency is approximately 400 microseconds on the nCUBE-2, compared to 5 milliseconds on the cluster. In addition to software overhead, the increased latency results from a context switches.

Finally, the cluster has faster processors than the hypercube: the sequential version of the split-merge program runs approximately 6 times faster on a lightly loaded workstation than on a single node in the nCUBE-2. While this difference benefits cluster applications, it also magnifies the disadvantage of the cluster in terms of communication latency.

4.2 Load Balancing Strategies

As described in Section 3, the nCUBE-2 version of the split-merge algorithm used dynamic load balancing to accommodate the variance in the Laguerre iteration routine. In the hypercube environment, load balancing was useful only in the distributed stages of the algorithm. However, traces of execution on the cluster indicated that the problem of load imbalance could also appear in the local stages, that is, in those stages involving no interprocessor communication. This behavior can be attributed to a PVM process relinquishing the processor to another user application on a particular workstation. When such an event occurs, the execution of the entire program is delayed until that PVM instance processes finishes its (local) share of the work.

This phenomenon can be observed in Figure 5, where every process is executing on a different workstation. Process 6 was assigned by PVM to a workstation occupied by an interactive user. Although the distributed stages (9-11) are reasonably well balanced, the execution of process 6 on the last local stage (8) delays the entire algorithm. This result illustrates that in a cluster of workstations load balancing may be critical to parts of the application where the workload was evenly distributed in an MPC implementation. In order to alleviate this problem in the split-merge algorithm, load balancing was also applied to the last several local stages. If all workstations have the same level of processor utilization, then there is no advantage in using load balancing in the local stages, as it introduces additional communication costs. However, if the load on the workstations is uneven, or the capacities of the workstations are uneven, then load balancing in the local stages reduces the overall execution time.

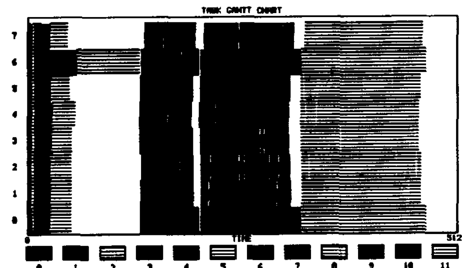


Figure 5. Example of load imbalance

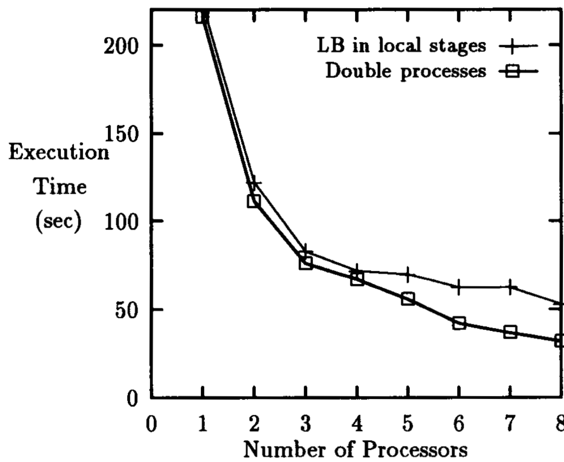
4.3 Masking Communication Costs

An approach that proved effective in masking the cost of communications on the cluster was to create more worker processes than there are workstations, which is possible because PVM can map several processes to the same processor. This approach reduced the negative effect of processes becoming blocked while waiting for communication to complete.

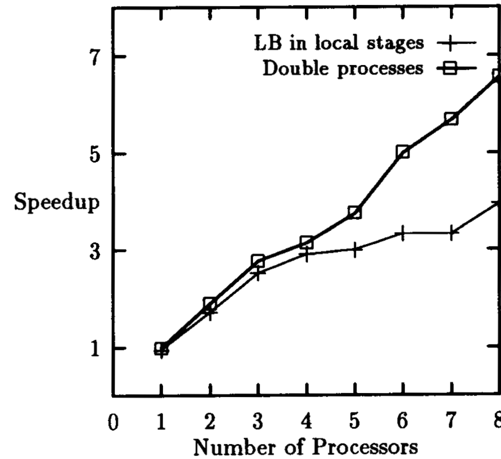
In the PVM implementation, this approach proved even more successful than load balancing in local stages, when all workstations in the cluster have a low level of utilization, as can be observed in Figure 6. Figures 6(a) and 6(b), respectively, plot the execution times and corresponding speedups when two processes were created and assigned to each workstation. These two strategies are useful for different problems. Load balancing in the early stages helps when a subset of the workstations has a higher level of utilization than the rest of the workstations in the cluster. Doubling the number of processes in each workstation helps to mask the cost of communications.

4.4 Comparison with the Hypercube Performance

Although the communication costs are substantially higher on the cluster environment, the greater computing power of the processors results in good performance. Since the split-merge algorithm is not particularly communication-intensive, it benefits substantially from this characteristic of the cluster. Therefore,



(a) execution times



(b) speedups

Figure 6. Comparison of load balancing approaches for a matrix or order 4096

when comparing cluster performance with that of the nCUBE-2, it is necessary to examine the *absolute* execution times and to consider the cost and flexibility of the systems.

As an example, Figure 7 plots the measured execution time of the distributed algorithm on a particular problem instance of size 2048, as executed on an nCUBE-2 and a lightly loaded cluster. Finding all the eigenvalues of the matrix required 127 seconds on a 8-node cube, 60 seconds on a 16-node cube, 31 seconds on a 32-node cube, and 16 seconds on a 64-node cube. On the cluster, the same problem required 77 seconds on one node, 21 seconds on 4 nodes, and 15 seconds on 8 nodes. The cluster times with 4 and 8 processors are competitive with the times of the nCUBE-2 with 32 and 64 processors, respectively.

In light of these results, and taking into consideration the cost factor (given university discounts on equipment, 16 Sparc-10s cost approximately the same as 16-node nCUBE-2), it can be argued that clusters of workstations provide a cost/effective alternative to MPCs for certain scientific applications, once the higher communication costs are accounted for. In this case, the use of two processes per node was effective in hiding communication latency. Improving performance by reducing broadcast times and using faster LANs is part of our ongoing research.

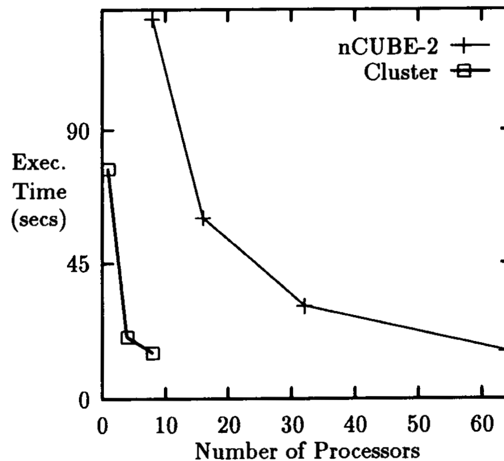
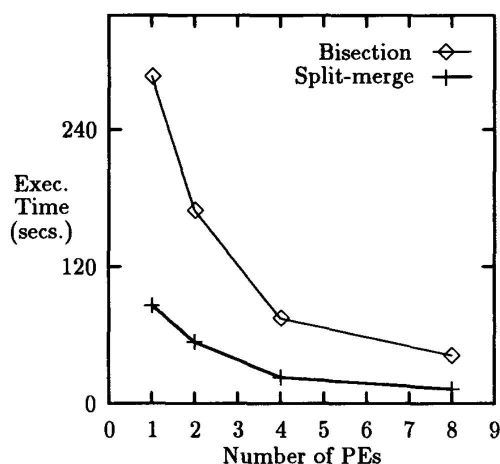


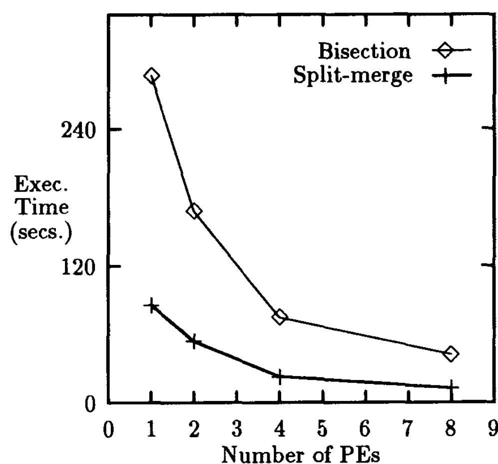
Figure 7. Execution time comparison of nCUBE-2 and cluster of workstations

5 Performance Study

The preceding discussion was based on experiments performed using random matrices. To verify the robustness of the distributed, parallel split-merge algorithm, several other types of input matrices, designed specifically to test the accuracy and speed of eigenvalue solvers, were used.



(a) Matrix with 2048 entries of type 1.



(b) Matrix with 2048 entries of type 2.

Figure 8. Execution times of cluster eigenvalue solvers for Type 1 and 2 matrices.

A comparison of the *sequential* version of the split-merge algorithm with other sequential algorithms for finding eigenvalues was conducted previously by Li and Zeng [7]. In that study, the other sequential algorithms included bisection/multisection(DSTEBZ in LAPACK), divide-and-conquer (TREEQL [4]), RFQR (Root-free QR, DSTERF from LAPACK) and QR (DSTEQR from LAPACK). Split-and-merge achieved the best accuracy, while RFQR was the fastest sequential algorithm.

In our study, the parallel version of split-merge was compared against a parallel version of the well-known method of bisection [9]. Several parallel versions of the bisection method have been developed to find eigenvalues of symmetric tridiagonal matrices. The one that was used for the comparison is geared towards maximum parallelism. First, the interval containing all the eigenvalues of the matrix is calculated. Once this interval has been calculated, the number maximum of iterations required to obtain an eigenvalue with the desired accuracy is calculated. Without describing the details here, every eigenvalue can be obtained independently from the others, so that the algorithm can be parallelized by distributing the eigenvalues among the available processors. Thus, bisection is very well suited for parallelization: once the initial data has been broadcast to the participating nodes, each node can work on its part of the problem without any communication with other nodes, except for reporting the final results to the initiating node. QR methods are difficult to parallelize when only the eigenvalues are sought [3]. Ipsen and Jessup [6] report that parallel bisection is faster than divide-and-conquer, hence the decision to use bisection in comparisons reported here.

Twelve standard types of input matrices were used in the experiments and are described in the technical report [13]. The results for matrices type 1 and 2 are discussed here. The following conventions are used in the description of the matrices: $\alpha_i, i = 1, ..., n$ represent the (main) diagonal entries and $\beta_i, i = 1, ..., n-1$ denote the offdiagonal entries. For matrices of type 1 (Toeplitz matrices), $\alpha_i = a$ and $\beta_i = b$, for all i. Matrices of type 2 have $\alpha_1 = a - b$, $\alpha_i = a$, for $i = 2, \ldots, n-1$, $\alpha_n = a + b$, and $\beta_j = b$, for $j = 1, \ldots, n-1$. The values of a and b were chosen to be 4 and 1, respectively, for all the matrices. The experiments were performed on matrices of size 128, 256, 512, 1024 and 2048 for types 1 through 7, size 128, 256 and 512 for types 8 through 12. Clusters of 1, 2, 4 and 8 workstations were used.

Figure 8 shows the execution times for matrix types 1 and 2 with 2048 entries. The results for types 3 through 12 are similar to those in Figure 8, and are omitted here for the sake of brevity; those results are available in [13]. The sequential version of the split-merge algorithm is significantly faster than the sequential version of bisection. In examining absolute execution times of the parallel algorithm, for matrices with 2048 entries, the execution times of the parallel version of split-merge on a cluster with 4 nodes are similar to the execution times of split-merge on a 32-node nCUBE-2, confirming the observations made pre-

viously for random matrices that the implementations on small clusters are competitive with those on a much larger nCUBE-2. Specifically, for both the bisection and the split-merge algorithms, a very small cluster of workstations represents a viable alternative to an nCUBE-2 with 32 nodes.

The experiments were generally conducted at times of little activity on the workstations, but as the workstations are in an open laboratory, it was not possible to guarantee exclusive access to the systems. It should also be noted that faster networks and better interfaces between the workstation and the networks are becoming available and they will reduce the communication overhead of the clusters. As communications latencies decrease, the advantage of split-merge will likely increase.

6 Conclusions and Future Work

The split-merge algorithm is a new, highly parallelizable method for finding eigenvalues. This paper described its first distributed parallel implementation on a workstation cluster. The work described in this paper differs from previous contributions in parallel eigensolvers by focusing on how to accommodate and exploit particular features of a distributed environment in order to maximize the performance. Specifically, we evaluated the effects on performance of dynamic load balancing and the use of multiple processes per node. The resulting implementations demonstrated performance that compares very well with other parallel and distributed methods.

Our continuing research on highly-parallel eigenvalue solvers and distributed parallel computing comprises several tasks. For larger clusters of workstations, it will be necessary to reduce communication costs, particularly the cost of broadcast. This will be accomplished on a regular Ethernet by incorporating IP multicast [14] into PVM. In the near future, the Ethernet in the cluster will be replaced with an ATM LAN that supports broadcast communication in hardware. Other environments and platforms will also be used to study the performance under different conditions.

References

1. H. T. Kung, R. Sansom, S. Schlick, P. Steenkiste, M. Arnould, F. Bitz, F. Christianson, E. Cooper, O. Menzilcioglu, D. Ombres, and B. Zill, "Network-based multicomputers: An emerging parallel architecture," in *Proceedings of Supercomputing '91*, pp. 664–673, 1991.
2. T.-Y. Li, H. Zhang, and X.-H. Sun, "Parallel homotopy algorithm for the symmetric tridiagonal eigenvalue problem," *SIAM Journal of Scientific and Statistical Computing*, vol. 12, pp. 469–487, May 1991.
3. P. Arbenz, K. Gates, and C. Sprenger, "A parallel implementation of the symmetric tridiagonal QR algorithm," in *Proceedings of the Fourth Symposium on the Frontiers of Massively Paralllel Computation*, IEEE CS Press, 1992.
4. J. J. Dongarra and D. C. Sorensen, "A fully parallel algorithm for the symmetric eigenvalue problem," *SIAM Journal of Scientific and Statistical Computing*, vol. 2, pp. 139–154, March 1987.
5. S. S. Lo, B. Philippe, and A. Sameh, "A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem," *SIAM Journal of Scientific and Statistical Computing*, vol. 2, pp. 155–165, March 1987.
6. I. C. F. Ipsen and E. R. Jessup, "Solving the symmetric tridiagonal eigenvalue problem on the hypercube," *SIAM Journal of Scientific and Statistical Computing*, vol. 11, pp. 203–229, March 1990.
7. T. Y. Li and Z. Zeng, "Laguerre's iteration in solving the symmetric tridiagonal eigenproblem - revisited," *SIAM Journal of Scientific Computing*, 1993. accepted to appear.
8. C. Trefftz, P. K. McKinley, T. Y. Li, and Z. Zeng, "A scalable eigenvalue solver for symmetric tridiagonal matrices," in *Proceedings of the sixth SIAM conference on Parallel Processing*, pp. 602–609, 1993.
9. G. H. Golub and C. F. Van Loan, *Matrix Computations*. Johns Hopkins University Press, 2nd. edition ed., 1990.
10. N. G. Shivaratri, P. Krueger, and M. Singhal, "Load distributing for locally distributed systems," *IEEE Computer*, vol. 25, pp. 33–44, December 1992.
11. V. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, December 1990.
12. J. Boyle, R. Butler, T. Disz, B. Glickfield, E. Lusk, and R. Overbeek, *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, 1987.
13. C. Trefftz, P. K. McKinley, T. Y. Li, and Z. Zeng, "A scalable eigenvalue solver for symmetric tridiagonal matrices," Tech. Rep. MSU-CPS-ACS-69, Michigan State University, 1992.
14. S. E. Deering and D. R. Cheriton, "Multicast routing in datagram internetwork and extended LANs," *ACM Transactions on Computer Systems*, vol. 8, pp. 85–110, May 1990.