A PARALLEL IMPLEMENTATION OF THE INVARIANT SUBSPACE DECOMPOSITION ALGORITHM FOR DENSE SYMMETRIC MATRICES*

STEVEN HUSS-LEDERMAN†,§, ANNA TSAO†,§, AND GUODONG ZHANG‡,§

Appeared in Proceedings, Sixth SIAM Conference on Parallel Processing for Scientific Computing, Norfolk, Virginia, March 22–24, 1993.

ABSTRACT. We give an overview of the Invariant Subspace Decomposition Algorithm for dense symmetric matrices (SYISDA) by first describing the algorithm, followed by a discussion of a parallel implementation of SYISDA on the Intel Delta. Our implementation utilizes an optimized parallel matrix multiplication implementation we have developed. Load balancing in the costly early stages of the algorithm is accomplished without redistribution of data between stages through the use of the block scattered decomposition. Computation of the invariant subspaces at each stage is done using a new tridiagonalization scheme due to Bischof and Sun.

1. INTRODUCTION

Computation of all the eigenvalues and eigenvectors of a dense symmetric matrix is an essential kernel in many applications. The ever-increasing computational power available from parallel computers offers the potential for solving much larger problems than could have been contemplated previously. Hardware scalability of parallel machines is frequently achieved by physically distributing the global memory across the machine; memory access is then handled via either an implicit or explicit message-passing communications protocol. The effects of memory latency and the internode distance, or "hops", are usually pivotal considerations in the design of algorithms for such machines. The prevalence of distributed memory machines has spurred very active investigation into algorithms which map more readily onto these architectures. Promising algorithms that have been investigated include bisection/multisection coupled with inverse iteration, Divide and Conquer, Jacobi methods, and homotopy methods [29, 23, 4, 26].

The work presented in this paper is part of the Parallel Research on Invariant Subspace Methods (PRISM) project, composed of researchers from Argonne National Laboratory, Supercomputing Research Center, University of California, Berkeley, and University of Kentucky, which is investigating scalable algorithms for solving eigenproblems. In the context of the current work, an algorithm is *scalable* if for fixed granularity, it is able to maintain efficiency as the number of processors increases.

We present preliminary results on parallel studies of the Symmetric Invariant Subspace Decomposition Algorithm (SYISDA). In particular, we present results from work-in-progress on an implementation of SYISDA for square meshes. SYISDA is a divide-and-conquer algorithm which derives most of its parallelism from performing matrix multiplication [3]. Recall the algorithm in [25].

---

*This paper is PRISM Working Note #9, available via anonymous ftp to `ftp.super.org` in the directory pub/prism.

†Supercomputing Research Center, 17100 Science Drive, Bowie, MD 20715

‡CONVEX Computer Corporation, 3000 Waterview Parkway, Richardson, TX 75083-3851. This research was done while author was at Department of Mathematics, University of Maryland, College Park, MD 20742

§This research is being partially supported by DARPA's Applied and Computational Mathematics Program, under contract DM28E04120. Access to the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium is being provided by NSF.

1991 *Mathematics Subject Classification.* 65F15.

Key words and phrases. eigenvalues, symmetric matrices, divide and conquer algorithm, invariant subspaces, parallel algorithm.

Typeset by $\mathcal{AMS}$-TEX

Symmetric Invariant Subspace Decomposition Algorithm (SYISDA).
Suppose that A is a $n \times n$ dense symmetric matrix. If eigenvectors are desired, initialize the eigenvector matrix Z to be the identity matrix.

1. *Compute upper and lower bounds on the spectrum of A and use these bounds to form $\ell(A)$, where ℓ is a linear function that maps the spectrum of A into $[0, 1]$ so that its mean eigenvalue is mapped to $1/2$.*
2. *Perform the iteration,*
$$\begin{aligned} A_0 &= \ell(A), \\ A_k &= \mathcal{P}_k(A_{k-1}), \qquad k = 1, 2, 3, \ldots, \end{aligned}$$
where the polynomials $\mathcal{P}_k, k = 1, 2, 3, \ldots$ are designed so that the eigenvalues of the sequence of iterates tend either toward 0 or 1. The iterations terminate when the sequence of matrices converges numerically.
3. *Compute the two independent subproblems of A, $A_1 = Q_1^t A Q_1$ and $A_2 = Q_2^t A Q_2$, where $Q = [\,Q_1 \quad Q_2\,]$ is an orthogonal matrix such that the columns of Q_1 and Q_2 are a basis for the range and null space of the iterated matrix, respectively. If eigenvectors are also desired, Z must be updated to ZQ.*

A_1 and A_2 can be solved totally independently and the algorithm applied recursively until all subproblems are diagonal matrices. The divide-and-conquer nature of the algorithm has a tree-like structure where the number of subproblems doubles at each level in the tree, resulting eventually in a diagonal matrix whose diagonal entries are the eigenvalues of A. Orthogonality in the eigenvectors is guaranteed due to the exclusive use of orthogonal transformations.

In previous sequential investigations, the polynomials $\mathcal{P}_k$, $k = 1, 2, 3, \ldots$, were all just the first incomplete Beta function, $\mathfrak{B}(x) = 3x^2 - 2x^3$. The use of a more specialized algorithm for the invariant subspace computation in Step 3 necessitated a slight modification of this strategy, which will be detailed in a future paper.

We believe that SYISDA is a potentially useful algorithm for large problems because of its scalable primitives. As discussed in [25], Step 2 consists entirely of matrix multiplication and dominates the cost of the sequential algorithm. Because SYISDA works with dense matrices, it is significantly more expensive from a computational complexity point of view than the widely-used QR algorithm. SYISDA is nonetheless a promising algorithm, since matrix multiplication is a preferred primitive on a wide variety of architectures [17, 24, 19]. For instance, on machines such as the Cray-2, Cray-YMP, and IBM RS6000/550, where optimized vendor-supplied BLAS are supplied, actual running times for SYISDA are usually within a factor of 3–6 of the symmetric QR algorithm, even when symmetry is not exploited. Although the amount of computation required in Step 3 is significantly smaller than for Step 2, effective parallelization of this step is critical, since matrix multiplication is highly scalable. The implementation of these two steps is discussed in Section 2.

Although the initial distributed implementation of SYISDA is being done on the Intel Touchstone Delta, our algorithmic goal is a portable distributed algorithm for mesh-based machines. In particular, we chose a modular design which achieves scalability through distributed versions of the dominant primitives, matrix multiplication and the divide. In this way, machine-dependent communications are relegated as much as possible to these primitives, localizing their impact. Efforts to attain performance can then be devoted to optimizing these functions on the target machine. This philosophy is analogous to the reliance of packages such as LAPACK [2] and ScaLAPACK [13] on optimized BLAS. General parallel design issues for SYISDA on square meshes are discussed in Section 3. In particular, we describe how good load balancing in the critical early stages of SYISDA can be achieved by spreading the data for each subproblem over the entire mesh. This is accomplished on square meshes using the block scattered decomposition [13] and eliminates the need for costly and complicated data movement between stages. In the late stages of the algorithm, when communication overheads become much more significant, a one-time global data redistribution is performed to finish the small amount of remaining work. Plans for future work are briefly described in Section 4.

2. Optimization of Primitives on the Intel Touchstone Delta

2.1. Description of Delta.

The Delta is Intel's successor to the iPSC family of parallel computers and the third in the Intel Touchstone line. The Delta resides at Caltech and is owned by the Concurrent Supercomputing Consortium. The Delta, unlike its hypercube predecessors, is a two-dimensional mesh, based on the i860 and employing a packetized wormhole (e-cube) routing message-passing protocol.

Four characteristics of the Delta influenced our choice of algorithm. First, contention and concurrency cause significant communication delays in many cases. *Contention* exists when two messages are competing for the same physical wire. *Concurrency* exists at a node when the node must deal with two different messages simultaneously. The Delta network is able to deal with modest levels of contention, but is greatly impacted by concurrency. Minimization of contention and concurrency was therefore an important design consideration. Second, the Delta cannot effectively overlap computation and communication when the communication is optimized because the i860 at each node must mediate all memory requests. The context-switching that occurs when attempting to overlap communication and computation can actually have an adverse effect on performance. Third, the Delta's separate message-passing chips (MRCs) and wormhole routing protocol reduce communication delays enough to be able to assume that "hops" are almost free. More details on Delta communications can be found in [27], [16], [28], and [21]. The final important feature of the Delta we exploited is that the assembly-coded version of DGEMM can sustain a peak rate of 36.5 Mflops/sec per node of a possible 40 Mflops/sec (peak rate for one addition and one multiplication) for sufficiently large matrices. The achievable rate declines significantly as the size of the matrices being multiplied decreases, making granularity considerations critical in the design.

2.2. Key primitives.

As noted before, the performance of SYISDA depends heavily on providing a scalable matrix multiplication code on the Delta. Our initial implementation does not exploit symmetry in performing matrix multiplication. Although existing codes provide good performance, our long-term goals necessitated development of a dense matrix multiplication code able to accommodate arbitrary rectangular mesh configurations and matrix dimensions. This in turn provided the motivation for a new generalization of the block scattered decomposition on non-square meshes that also allowed for good performance in distributed matrix multiplication. This generalization, virtual two-dimensional panelled torus wrap, is discussed in another paper in these proceedings [22].

We implemented a variant of Broadcast-Multiply-Roll [17] in C using communication primitives highly suited to the Delta. In particular, since the Delta does not effectively overlap communication and computation, the algorithm we chose is highly synchronous. In addition, the algorithm maintains a large granularity so that the single node assembly-coded matrix multiplication could be exploited. Our algorithm is completely general, able to deal with arbitrary rectangular mesh and matrix dimensions, and has achieved parallel efficiency of 86% with overall peak performance in excess of 8 Gflops on 256 nodes for an 8800×8800 double precision matrix. Our code can deal with a variety of different blocking schemes, by means of a user-passed function providing block size information. More detailed information about our implementation is given in [22]. The code was developed with square matrices in mind and performs best when the matrix elements are spread out as uniformly as possible. Optimal load balancing occurs when each node has the same amount of data for each matrix. The implementation guarantees that all matrix operands end up in place.

Bischof and Sun [9, 10, 8] have developed a new method for accomplishing Step 3. The application of the sequence of polynomials $\mathcal{P}_k$ in Step 2 produces a matrix M all of whose eigenvalues are either zero or one. To decompose the eigenproblem for A into two smaller ones, we must compute the range and null spaces of M. This is usually achieved with a so-called "rank-revealing orthogonal factorization" [7, 11, 12]; the "QR-factorization with column pivoting" [18], used in our previously implemented sequential code, is the most commonly implemented strategy [14, 1]. For computers with a memory hierarchy and distributed-memory machines, variations of this strategy have been devised [6, 5] which limit the adverse effect of the data-dependent column exchanges on the data reference locality and synchronization costs. One numerical drawback of the QR factorization with column pivoting is that, although it has proven to be reliable in practice, it is not guaranteed to be foolproof for computing the range and null space of M in Step 2.

The invariant subspace computation problem arising in the SYISDA context has a special structure that can be exploited very advantageously, resulting in an algorithm having a very regular and predictable data and communication flow pattern. In particular, the eigendecomposition of M can be obtained by first tridiagonalizing M and then solving at most $n/2$ independent 2×2 eigenvalue problems. The matrix of eigenvectors of M corresponding to the eigenvalue one is a basis for the range space of M and the matrix of eigenvectors of M corresponding to the eigenvalue zero is a basis for the null space of M. Numerically, M does not of course have eigenvalues exactly 0 and 1, but does not prove to be a problem. This procedure will be described in more detail in a forthcoming paper.

The reduction of M to T and the accumulation of the transformation matrix Q dominate the invariant subspace computation. Several parallel tridiagonalization implementations have been suggested in the literature based either on the standard Householder tridiagonalization procedure (e.g., [20]) or a block version of this algorithm (e.g., [15]). Bischof and Sun, on the other hand, are developing a parallel implementation for square meshes of the band reduction paradigm described in [10]. The reduction of a full matrix to tridiagonal form is a special case covered by this paradigm. This approach is described in more detail in [8] in these proceedings.

3. Parallel Issues for SYISDA

3.1. The early stages.

For large problems, the early stages of SYISDA are dominated by the computation time for performing the matrix multiplications in Step 2 and the divide in Step 3. The remaining critical design issue is load balancing of multiple subproblems. In this section, we discuss the approaches we considered for the early stages of SYISDA. Throughout this section, we assume that the computation is being done on a $p \times p$ mesh.

Because we chose a two-dimensional matrix multiplication algorithm, the two data layouts considered here are two-dimensional contiguous blocking and the block scattered decomposition (e.g., [22]). Let us briefly review both data layout strategies on a $p \times p$ mesh. Let ω be the block size. For simplicity, assume that ω divides the matrix dimension, n. We "panel" the matrix M in both the row and column dimensions into panels of width ω, resulting in blocks of size $\omega \times \omega$. Panels in each matrix dimension are torus-wrapped in the corresponding dimensions of the mesh, i.e., panel i in the row dimension ends up in row $i \mod p$ of the mesh and panel j in the column dimension ends up in column $j \mod p$ of the mesh. For square meshes, contiguous blocking refers to the case where $\omega = n/p$. If $\omega < n/p$, ω is generally chosen to be small relative to n and optimal performance in performing dense matrix multiplication is achieved when $\omega = 1$, but larger block sizes are desirable for blocked operations in the divide step.

Suppose the nodes are numbered so that the node in row i and column j is assigned the number $ip + j$. Figure 1 compares contiguous blocking and the block scattered decomposition when $n = 60$ and $p = 3$. Suppose that SYISDA divides A into two subproblems of sizes 20 and 40, respectively. The subproblems A_1 and A_2 produced by Step 3 are enclosed in the dark boxes. In the contiguous block case, $\omega = 20$, whereas in the block scattered case, $\omega = 5$. Note that with contiguous blocking, not all nodes will have data for the resulting subproblems, whereas in the block scattered case, all nodes will contain data for both subproblems for sufficiently large subproblems.

The first approach to load balancing is to try to solve multiple subproblems in parallel by means of a dynamic load balancing scheme that repartitions the mesh between stages and assigns different subproblems to distinct submeshes. There are two main reasons why this approach is unattractive for large subproblems. First, an effective dynamic strategy requires the data associated with each subproblem to be moved between each stage. This data movement is complicated, potentially very costly when the subproblems are large due to the effects of contention and concurrency, and as we shall discuss later in this section, somewhat unnecessary. In Figure 2(a), we compare the sizes of the submeshes containing data for A_1 in both blocking schemes as well as that required for "perfect" dynamic load balancing under our idealized assumptions, expressed as percentage of the total number of nodes in the mesh. We see that both blocking schemes would

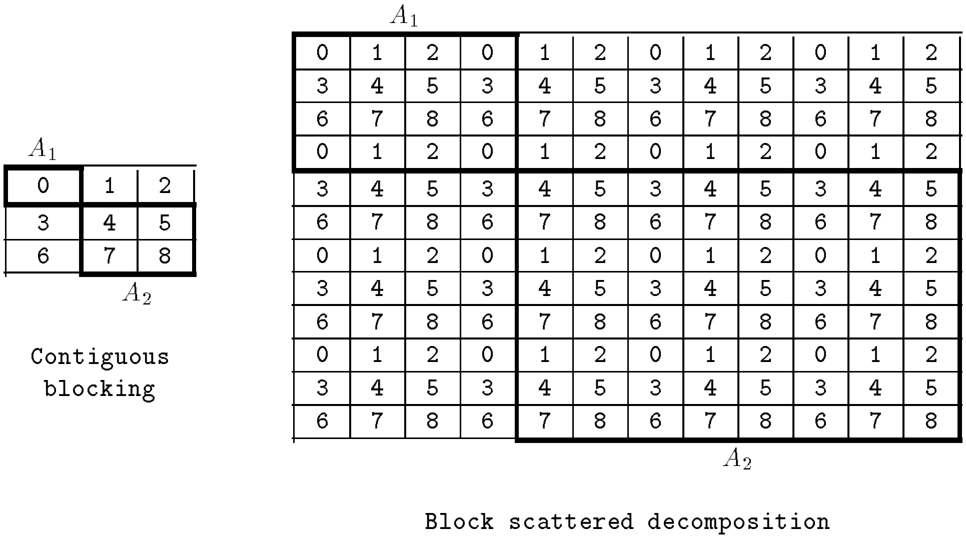

Figure 1. Data layout examples

require considerable global data movement achieve the right submesh allocations for "perfect" dynamic load balancing. This data movement can involve high levels of both contention and concurrency.

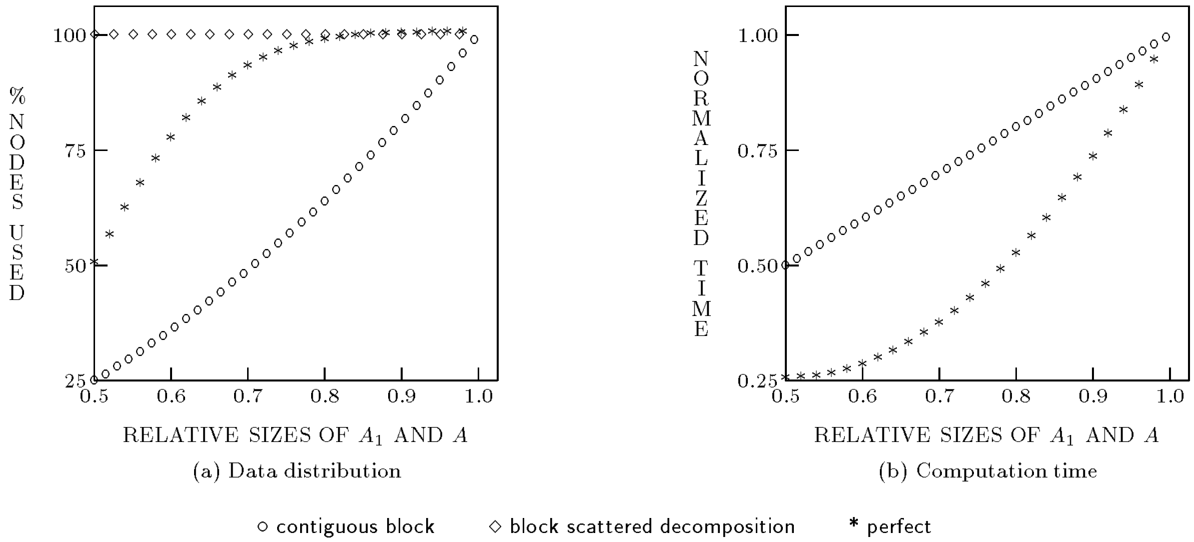

Figure 2

The second, more fundamental reason is that there is no way to predict *a priori* how much work will be required for remaining subproblems, since the subproblem sizes and number of iterations required in Step 2 depend on the distribution of the eigenvalues. As an extreme example, when the spectral bounds in Step 1 indicate that a particular subproblem has clustered eigenvalues [25], no iterations at all are required. In the early stages of SYISDA, the run time differences caused by poor load balancing could have an extremely adverse effect on performance, making dynamic load balancing problematic.

The other approach to load balancing is not to move the data at all between the early stages. We shall call this the "static" load balancing scheme. The choice of data layout in the static scheme is an important consideration. Figure 1 demonstrates that contiguous blocking will result in some nodes being idle during

Steps 1 and 2 for subsequent divides, but all subproblems can be done in parallel. The granularity of the local computation is the same as that of the original subproblem. On the other hand, in the block scattered case, if the subproblems are sufficiently large, each subproblem is still spread out over the entire mesh as uniformly as the block size allows. The avoidance of data movement between divides is achieved at the cost of more communication during individual matrix multiplications, having to solve subproblems sequentially, and finer-grained local computation. However, this approach guarantees near-perfect load balancing in the costly early stages.

We now present a very crude comparison of the dynamic and the fixed load balancing schemes, considering both contiguous blocking and the block scattered decomposition . To simplify the analysis, assume that for a subproblem of size m, the amount of sequential computation required to finish that problem is Cm^3, where C does not vary with problem size. Let P be the number of nodes. Now suppose that a problem, A, of dimension n divides into two subproblems, A_1 and A_2 of dimensions m and $n-m$, respectively. Without loss of generality, assume $m \geq n-m$. Let r be the fraction of the entire mesh used to solve A_1 and s be the ratio of the dimensions of A_1 and A. For the moment, we ignore differences in the communication cost for matrix multiplication, the cost of redistributing the data between levels, and the effects of computational granularity and assume perfect speedups. In the dynamic load balancing scheme, "perfect" load balancing is achieved if the fraction of the mesh used for solving A_1 equals the fraction of the total remaining computation time that is spent solving A_1, i.e., r and s satisfy

$$r = \frac{C(sn)^3}{C(sn)^3 + C((1-s)n)^3} = \frac{s^3}{1-3s+3s^2}. \tag{1}$$

In this case, if P is the total number of processors and $r \leq s^3/(1-3s+3s^2)$, then the total computation time for both subproblems, T_d, is just the time needed to solve A_1, i.e.,

$$T_d = \frac{C(sn)^3}{rP} = \frac{Cs^3}{r}\left(\frac{n^3}{P}\right). \tag{2}$$

Let T_c and T_s be the times required to solve A_1 and A_2 using the static load balancing strategy with contiguous blocking and the block scattered decomposition, respectively. It is clear that $T_c \geq T_d$, with equality (hopefully) rarely occurring. On the other hand, with the static block scattered decomposition strategy, the total remaining computation time is the sum of the times to solve both A_1 and A_2, i.e.,

$$T_c = \frac{C(sn)^3}{P} + \frac{C((1-s)n)^3}{P} = (1-3s+3s^2)\left(\frac{Cn^3}{P}\right). \tag{3}$$

Note that $T_c = T_d$ when r and s satisfy (1), i.e., the block scattered decomposition gives the same performance as a "perfect" dynamic load balancing scheme, under our assumptions.

Figure 2(b) shows the total time it would take to divide A_1 and A_2 for the "perfect" dynamic load balancing scheme versus the time required for static load balancing scheme using contiguous blocking. It is clear that there can be a severe penalty for the poor load balancing that can results from the static load balancing scheme using contiguous blocking, whereas the time required for the static load balancing scheme using the block scattered decomposition is the same as in the "perfect" scheme.

We therefore chose the static load balancing approach because under our idealized assumptions, it provides comparable performance to the perfect dynamic scheme as well being insensitive to differences in the time required to solve subsequent subproblems. Since the granularity of the local computation in the static approach chosen is usually smaller than for the "perfect" dynamic load balancing case, the actual time required will generally be larger than in the "perfect" case. However, performance differences between the two schemes should be small for large problems. After a few stages of SYISDA, the amount of data to be moved would be small compared to the size of the original problem, so it may become desirable to perform some data redistribution to assign different subproblems to distinct submeshes. This action would not be taken for the sake of load balancing, but rather because of diminishing performance caused by leaving small subproblems across the entire mesh. On the other hand, very little work will remain, so even if a significant improvement in performance results from doing this, it may not be worthwhile from the perspective of the overall performance of SYISDA. We will be studying this issue in more detail in the future.

When multiple subproblems exist, the solutions of the different subproblems could be interleaved so that a single stage of the matrix multiplication is done for all subproblems at once, reducing the effects of communication startup. For large subproblems, where most of the computation occurs and communication startup costs are not expected to be the main concern, this strategy may only provide minimal performance gains, while making the algorithm less portable. More investigation of this scheme needs to be done.

3.2. The end game.

As the subproblem sizes decrease, there is a point, probably machine-dependent, at which communications overhead makes it impractical to continue solving each subproblem over the entire mesh. The amount of work required to find the eigensolutions of the remaining subproblems is very small, but the cost of the update of Z, whose leading dimension is still n, is still quite large. Any number of schemes could work well in finishing up the problem. We chose a scheme that still exploits parallelism in the update of Z, but solves each individual subproblems on a single node. We elected to redistribute the columns of Z in order to coarsen the granularity of the update. First, we assign different subproblems to different nodes in such a way as to reduce contention in the data redistribution stage due to the e-cube protocol by assigning subproblems along diagonals of the mesh. This strategy is somewhat tailored towards the Delta, but because such small subproblems are involved, the impact of machine-dependent behaviors is expected to be minimal.

The columns of Z corresponding to the invariant subspace of each subproblem are moved into the same processor column as the subproblem, but are left spread out down that processor column, with the original two-dimensional panelling. After the data redistribution, a symmetric permutation of the subproblem (the permutation induced by the block scattering) is solved on its assigned node. The eigenvector matrix for the subproblem is then broadcast down the processor column to be multiplied by the invariant subspace matrix, where the updates are done sequentially over the subproblems in each column of the mesh. The eigenvectors and their associated eigenvalues are then redistributed over the mesh to have the same column (and row) panelling as the original matrix.

4. Future Work

Future work includes generalization of the entire algorithm to arbitrary rectangular meshes and the study of the dynamic load balancing approach described in Section 3.1. We intend to compare the performance of our algorithm to other distributed dense symmetric eigensolvers and study our algorithm on other machines such as the Intel Paragon.

5. Acknowledgements

The authors would like to thank Rik Littlefield (PNL), and Greg Regnier, David Scott, David Payne, Denise Ecklund, Ellen Deleganes, and Jerry Baugh (Intel Supercomputing Systems Division) for valuable discussions about the Intel Delta. We would also like thank the staff at the Caltech Concurrent Supercomputing Facilities for their cheerful and timely support.

The authors would particularly like to thank Chris Bischof and Xiaobai Sun for their collaborative efforts and valuable contributions to this work, as well as Elaine Jacobson for many helpful discussions during the course of this work.

References

1. Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, & D. Sorensen, *LAPACK: A portable linear algebra library for high-performance computers*, Proceedings, Supercomputing '90, IEEE Computer Society Press, Los Alamitos, California, 1990, pp. 2–11.
2. Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, & D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, 1992.
3. Auslander, L. & A. Tsao, *On parallelizable eigensolvers*, Adv. Appl. Math. **13** (1992), 253–261, (also appeared as Technical Report SRC-TR-91-028, Supercomputing Research Center).

4. Berry, M. & A. Sameh, *Parallel algorithms for the singular value and dense symmetric eigenvalue problem*, CSRD (1988), no. 761.
5. Bischof, C. H., *A block QR factorization algorithm using restricted pivoting.*, Proceedings SUPERCOMPUTING '89, ACM Press, Baltimore, Md., 1989, pp. 248–256.
6. Bischof, C., *A parallel QR factorization with controlled local pivoting*, SIAM J. Sci. Stat. Comput. **12** (1991), 36–57.
7. Bischof, C. H. & P. C. Hansen, *A block algorithm for computing rank-revealing QR factorizations*, also MCS-P251-0791, Numerical Algorithms **2** (1992), no. 3-4, 371–392.
8. Bischof, C., M. Marques, & X. Sun, *Parallel bandreduction and tridiagonalization*, Proceedings, Sixth SIAM Conference on Parallel Processing for Scientific Computing (Norfolk, Virginia, March 22-24, 1993) (R. F. Sincovec, eds.), SIAM, Philadelphia, 1993, (also PRISM Working Note #8).
9. Bischof, C. & X. Sun, *A divide-and-conquer method for tridiagonalizing symmetric matrices with repeated eigenvalues*, (also PRISM working note #1), Argonne National Laboratory (1992).
10. Bischof, C. & X. Sun, *A framework for symmetric band reduction and tridiagonalization*, Preprint MCS-P298-0392, Argonne National Laboratory (1992), (also PRISM Working Note #3).
11. Chan, T. F. & P. C. Hansen, *Some applications of the rank revealing QR factorization*, SIAM Journal on Scientific and Statistical Computing **13** (1992), no. 3, 727–741.
12. Chandrasekaran, S. & I. Ipsen, *On rank-revealing QR factorizations*, Technical Report YALEU/DCS/RR-880, Yale University, Department of Computer Science, 1991.
13. Choi, J., J. J. Dongarra, R. Pozo, & D. W. Walker, *ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers*, Proceedings, Fourth Symposium on the Frontiers of Massively Parallel Computation (McLean, Virginia, October 19–21, 1992), IEEE Computer Society Press, Los Alamitos, California, 1992, pp. 120–127.
14. Dongarra, J., C. B. Moler, J. R. Bunch, & G. W. Stewart, *LINPACK User's Guide*, SIAM, Philadelphia, 1979.
15. Dongarra, J. & R. van de Geijn, *Reduction to condensed form for the eigenvalue problem on distributed-memory architectures*, Technical Report ORNL/TM-12006, Oak Ridge National Laboratory, Engineering Physics and Mathematics Division (January 1992).
16. Dunigan, T. H., *Communication Performance of the Intel Touchstone Delta Mesh*, Technical Report ORNL/TM-11983, Oak Ridge National Laboratory (January, 1992).
17. Fox, G., S. Otto, & A. Hey, *Matrix algorithms on a hypercube I: matrix multiplication*, Parallel Computing **4** (1987), 17–31.
18. Golub, G. H., *Numerical methods for solving linear least squares problems*, Numerische Mathematik **7** (1965), 206–216.
19. Golub, G. & C. F. Van Loan, *Matrix Computations, 2nd ed.*, Johns Hopkins University Press, Baltimore, 1989.
20. Hendrikson, B. & D. Womble, *The torus-wrap mapping for dense matrix calculations on massively parallel computers*, SAND92-0792, Sandia National Laboratories (1992).
21. Huss-Lederman, S., E. M. Jacobson, A. Tsao, & G. Zhang, *Optimizing Communication Primitives on the Intel Touchstone Delta*, Technical Note, Supercomputing Research Center (1993) (to appear).
22. Huss-Lederman, S., E. M. Jacobson, A. Tsao, & G. Zhang, *Matrix Multiplication on the Intel Touchstone Delta*, Proceedings, Sixth SIAM Conference on Parallel Processing for Scientific Computing (Norfolk, Virginia, March 22-24, 1993) (R. F. Sincovec, eds.), SIAM, Philadelphia, 1993, (summary of results also appeared in FY 1991–1992 Annual Report of the The Concurrent Supercomputing Consortium; expanded version to appear as Technical Report, Supercomputing Research Center, 1993; also PRISM Working Note #7).
23. Ipsen, I. & E. Jessup, *Solving the symmetric tridiagonal eigenvalue problem on the hypercube*, Tech. Rep. RR-548, Yale University (1987).
24. Johnsson, S. L. & C.-T. Ho, *Matrix multiplication on boolean cubes using generic communication primitives*, Parallel Processing and Medium Scale Multiprocessors, SIAM, Philadelphia, 1987.
25. Lederman, S., A. Tsao, & T. Turnbull, *A parallelizable eigensolver for real diagonalizable matrices with real eigenvalues*, Technical Report TR-91-042, Supercomputing Research Center (1991).
26. Li, T.-Y., H. Zhang, & X.-H. Sun, *Parallel homotopy algorithm for symmetric tridiagonal eigenvalue problems*, SIAM J. Sci. Stat. Comput. **12** (1991), no. 3, 469–87.
27. Regnier, G., *Intel Touchstone Delta Message Passing Performance*, Intel Supercomputing Systems Division (August, 1991), preprint.
28. Regnier, G., *Delta Message Passing Protocol*, presentation overheads, Proceedings, First Intel Delta Applications Workshop, CCSF-14-92 (February, 1992), Caltech Concurrent Supercomputing Facilities, Pasadena, California, 1992, pp. 173–178.
29. Schreiber, R., *Solving eigenvalue and singular value problems on an undersized systolic array*, SIAM J. Sci. Stat. Comput. **7** (1986), no. 2, 441–451.