The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers

J. Demmel* K. Stanley†

Abstract

We discuss timing and performance modeling of a routine to find all the eigenvalues and eigenvectors of a dense symmetric matrix on distributed memory computers. The routine, `PDSYEVX`, is part of the `ScaLAPACK` library. It is based on bisection and inverse iteration, but is not designed to guarantee orthogonality of eigenvectors in the presence of clustered eigenvalues. We use our validated performance model to conclude that `PDSYEVX` is very efficient for large enough problem sizes, nearly independently of input and output data layouts. However, efficiency will be low if interprocessor communication is too slow, such as on conventional workstation networks, or if per processor memory is too small, such as on the Intel Gamma. Modeling also helps us choose the appropriate algorithm to deal with clusters.

1 Summary

There are many algorithms for solving the dense symmetric eigenproblem[11]. Most of them consist of 3 steps: 1) reduce the dense matrix to tridiagonal form, 2) find the eigenvalues and eigenvectors of the tridiagonal matrix, and 3) back-transform the eigenvectors The `ScaLAPACK`[9] routine, `PDSYEVX`, uses bisection and inverse iteration for step 2).

Our ultimate goal is not just to produce a scalable routine, but one where step 2) is not the bottleneck. Among the many designs possibilities for step 2), we have so far chosen two. The choice in `PDSYEVX` is fast, but will not guarantee orthogonal eigenvectors if there are large clusters of narrowly separated eigenvalues. The choice in `PDSYEV`, which we do not discuss in detail, guarantees orthogonality but is slower than steps 1 and 3. One goal of performance modeling is to predict the performance of these and other step 2 designs, to help design an algorithm which is simultaneously fast and guarantees orthogonality. We discuss these design alternatives briefly at the end.

In addition, we use our performance modeling of `PDSYEVX` to understand its performance on existing computers, find bottlenecks and improve its performance, and predict its performance on new platforms. Our model, which depends on 6 machine parameters and the problem size n, predicts the performance on the CM-5 without vector units and Intel Gamma to within 10%-30% for all but very small n. It correctly predicts that we will reach near perfect parallel efficiency for large enough problems, which happens on the CM-5,

*Computer Science Division and Mathematics Dept, University of California, Berkeley CA 94708. The authors acknowledge NSF Infrastructure Grant CDA-8722788 as well as NSF grant ASC 9005933, ARPA Grant DM28E04120 via a subcontract from Argonne National Labs, and ARPA Grant DAAL003-91-C-0047 via a subcontract from the University of Tennessee.

†Computer Science Division, University of California, Berkeley CA 94708. The author acknowledges the National Science Foundation for his graduate student fellowship

Model Parameter	Description	Performance limited by	measured values μs		measured by
			CM5 w/o VUs	Gamma	
$\tau_{\texttt{DGEMM}}$	BLAS3	peak flop rate	1/3.0	35.	Stanley
$\tau_{\texttt{DGEMV}}$	BLAS2	main memory	1/2.0	25.	Stanley
$\tau_{\div}$	Divide		5.2	1.3	Stanley
τ_{lat}	message latency	comm. software	150	173	Whaley[13]
τ_{band}	bandwidth^{-1}	comm. hardware	1.62	2.86	Whaley[13]

Machine parameters used to model PDSYEVX

but that the Gamma memory is not large enough to solve such large problems. It also predicts low efficiency when running on a workstation network with high latency and low bandwidth. Finally, the model helped identify bottlenecks on the Gamma from using a slow divide instruction and slow random number generation, which let us speedup bisection and inverse iteration by a factor of nearly 4.

2 PDSYEVX

PDSYEVX is built using the BLAS[8] Basic Linear Algebra Subroutines, the PBBLAS[4] Parallel Block Basic Linear Algebra Subroutines, and the BLACS[13], Basic Linear Algebra Communication Subroutines. The tridiagonal reduction was written by Jaeyoung Choi [3]. Step 2 is broken into two parts, bisection and inverse iteration, parts of which were written by Inderjit Dhillon [5]. Both bisection and inverse iteration do $O(1)$ communication, with each processor responsible for a subset of eigenvalues and eigenvectors. Gram-Schmidt reorthogonalization of the eigenvectors is only performed within a single processor. Hence, if a cluster of eigenvalues is too large to fit on a single processor, orthogonal eigenvectors are not guaranteed. Details of PDSYEVX will appear in a future report.

3 Method

The performance depends on how much of the spectrum is required, whether eigenvectors are desired, how much accuracy is required, the matrix size n and distribution of eigenvalues, the machine load, the algorithm and its implementation, the number of processors p, the data layout, and performance characteristics of the underlying hardware and software.

Here we assume that all eigenvalues and all eigenvectors are needed to full accuracy, and that the machine is lightly loaded. Except for the discussion in section 5, we assume that clusters are relatively small, i.e. $< (\frac{n^2}{p})$. As discussed in section 5, we expect the cost of redistributing the data upon input and output to be less than 5% of the total time spent. Therefore, we model only the data layout which obtains the best performance, i.e. square or nearly square, with a block size just large enough to allow acceptable DGEMM (BLAS 3 matrix-matrix multiply) and DGEMV (BLAS 2 matrix-vector multiply) performance.

This allows us to model the performance of PDSYEVX using just five measured machine parameters, listed in table 1, counting the operations corresponding to each parameter. The BLACS timings were performed by Whaley[13]. The rest we performed ourselves. We validate the model against the actual running time, both of the subparts and the end to end running time. Our method is iterative. We learn the most when our predictions do not match measured times.

The performance of tridiagonalization and back-transformation is limited by the cost of

Task	Computation Costs			Communication Costs	
	DGEMM	DGEMV	divide cost	latency	bandwidth^{-1}
Reduction to Tridiagonal Form	$\frac{2}{3}\frac{n^3}{p}\tau_{\texttt{DGEMM}}$	$\frac{2}{3}\frac{n^3}{p}\tau_{\texttt{DGEMV}}$		$21n\tau_{lat}\lg p$	$5\frac{n^2}{\sqrt{p}}\tau_{band}\lg p$
Bisection		$120\frac{n^2}{p}\tau_{\texttt{DGEMV}}$	$60\frac{n^2}{p}\tau_{\div}$		
Inverse Iteration		$400\frac{n^2}{p}\tau_{\texttt{DGEMV}}$	$11\frac{n^2}{p}\tau_{\div}$		$4\frac{n^2}{p}\tau_{band}$
Back-transformation	$2\frac{n^3}{p}\tau_{\texttt{DGEMM}}$				$2\frac{n^2}{\sqrt{p}}\tau_{band}\lg p$

Total: $\frac{8}{3}\frac{n^3}{p}\tau_{\texttt{DGEMM}} + (\frac{2}{3}\frac{n^3}{p} + 520\frac{n^2}{p})\tau_{\texttt{DGEMV}} + 71\frac{n^2}{p}\tau_{\div} + 21n\tau_{lat}\lg p + 7\frac{n^2\lg p}{\sqrt{p}}\tau_{band} + 4\frac{n^2}{p}\tau_{band}$

PDSYEVX *performance model based on operation counts*

the calls to the BLAS and BLACS. The time for DGEMM will be modeled as $\tau_{\texttt{DGEMM}}mnk$ where m, n and k are the input matrix dimensions. Likewise, DGEMV time is modeled as $\tau_{\texttt{DGEMV}}mn$. The cost of a BLACS broadcast is modeled as $(\tau_{lat} + msg\_size\tau_{band})\lg p$

Other models[7] include an $O(n)$ initiation cost for DGEMM and DGEMV, and a $O(\frac{n^2}{\sqrt{p}})$ term representing the number of extra flops that are performed because of blocking. We do not include the $O(n)$ initiation cost for DGEMM and DGEMV because we believe that this cost will always be substantially smaller than the message latency cost. Likewise we omit the $O(\frac{n^2}{\sqrt{p}})$ term representing the extra flops because we believe that it will always be substantially smaller than the communication bandwidth cost.

We are not yet able to predict the performance of bisection and inverse iteration satisfactorily, because it is very compiler dependent. So, the numbers in table 2 are empirical, we do not expect this model to predict bisection and inverse iteration well on other architectures.

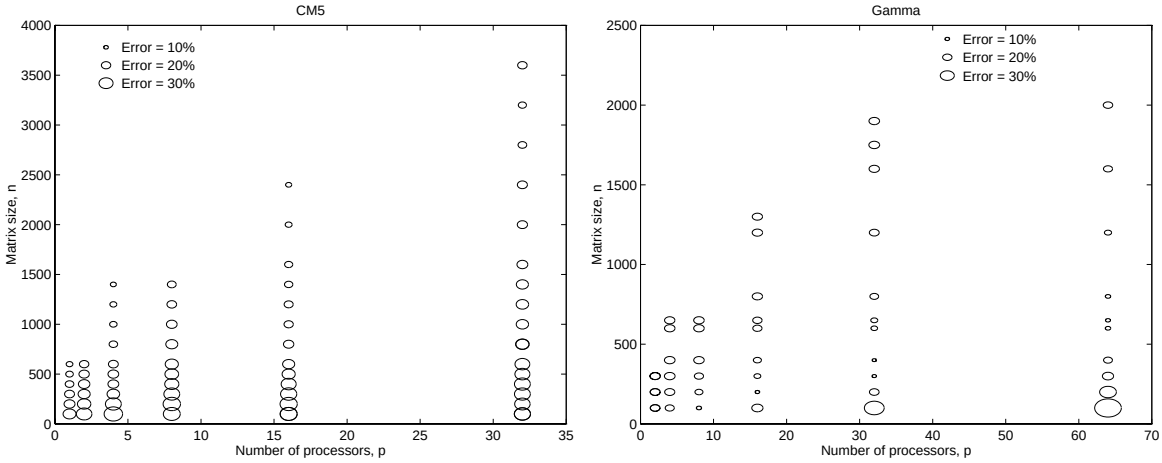
4 Validation

Our confidence in our models for reduction to tridiagonal form and back-transformation is based both on careful counting of flops and communication in the critical path, and on comparisons with measured data. Figure 1 presents our validation data. An ellipse located at coordinates (p, n) in the figure indicates a test with matrix dimension n run on p processors. The size of the ellipse is proportional to the error in the running time prediction. More precisely, it is proportional to the sum of the absolute values of the prediction errors in each of the four parts of PDSYEVX, divided by the actual end-to-end running time. This shows that not only is the full model accurately predict the total running time, but it also accurately predicts each of the four parts. The actual end-to-end prediction error is smaller in most cases because the errors in the four parts tend to cancel.

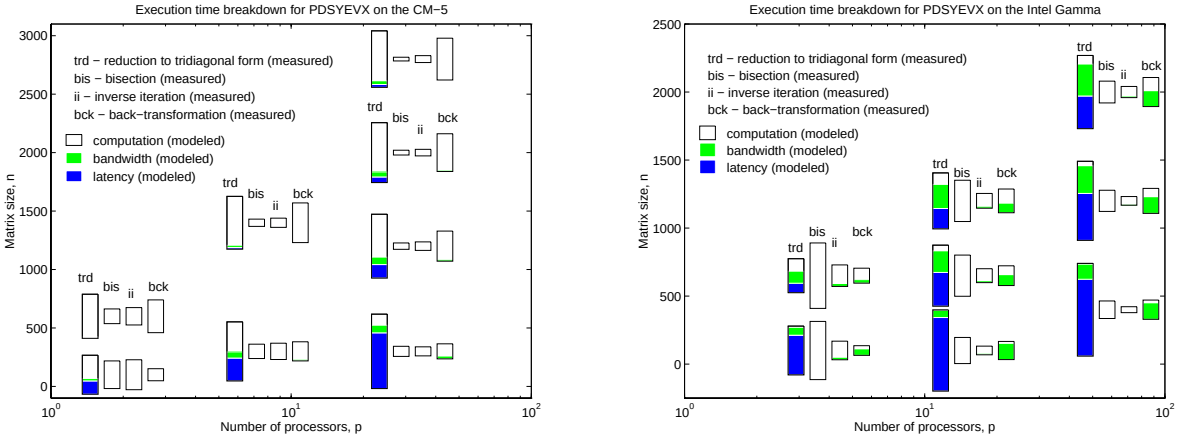
Figure 2 shows how time is distributed among computation, latency and bandwidth for all 4 parts of PDSYEVX, for varying n and p. The 4 vertical bars for each (p, n) correspond to tridiagonalization, bisection, inverse iteration, and back-transformation, respectively, and their heights add up to 1 unit, representing the total running time for that (p, n).

5 Conclusions

High Efficiency for high n^2/p. The largest terms in the timing formula in Table 2 are proportional to n^3/p, and represent the time of the serial algorithm for steps 1 and 3,



Sum of the absolute errors of the four parts of `PDSYEVX`*/ total time*



`PDSYEVX` *execution time breakdown*

divided by p. Since these dominate the other terms for large n^2/p (the data stored per processor), the algorithm is scalable, provided memory per processor is kept large. The largest problem that fits on the 32 processor CM-5 with 32 Megabytes of memory per node was $n = 2800$, which ran at 2.2 Mflops per node, an efficiency of 80%.

PDSYEVX will not tolerate existing network of workstations latency. The combination of high latency and a ring topology means that `PDSYEVX` will not work efficiently on existing implementations of `PVM`[1] on FDDI or Ethernetnetworks. Existing implementations of `PVM` have latencies around 1 to 5 milliseconds, 6 to 50 times higher than the latencies of the Gamma and CM-5. The ring topology changes the latency cost from $O(n \lg p)$ to $O(np)$ because concurrency between messages is not supported. The combination of these two factors will make the latency cost the dominant cost unless new workstations have hundreds of Megabytes of memory.

Need new algorithms to deal with large clusters. There are a large number of alternative algorithms for this problem [6]. We wish to avoid including full reorthogonalization, as in the serial `LAPACK` code `DSTEIN`, because this could increase both floating point and communication from $O(n^2)$ to $O(n^3)$ in the presense of large clusters. An alternative is `PDSYEV` in which QR is performed by each processor redundantly performing the

$O(n^2)$ effort of finding the shifts and performing $1/p^{th}$ of the $O(n^3)$ work of updating Q. `PDSYEV` will guarantee orthogonality and because it has $O(1)$ communication, it will scale well though it will be several times slower than `PDSYEVX`. Another possibility, which is much harder to program on a parallel machine, is Cuppen's divide and conquer routine, as modified by Eisenstat, Gu, Li and Rutter [12]. One attractive possibility is "intelligent brute force", or using simulated multiple precision arithmetic on clusters, which may require no new communication. Another possibility is spectral divide and conquer as in Tsao et al.[2] We are still studying the tradeoffs.

We are not reaching asymptotic speed on the Intel Gamma. Using a simple asymptotic model, we find that we would need 168 Mbytes per node to achieve 50% of the asymptotic speed, $\lim_{n\to\infty} \frac{time_{\texttt{PDSYEVX}}}{n^3/p}$, on a 32 processor Gamma. By contrast, we achieve 50% of the asymptotic speed with only 4.25 Mbytes per node on the CM-5. This is illustrated in figure 2. For $n = 2800$ on 32 processors of the CM-5, the vast majority of the time is spent in the computational parts of reduction to tridiagonal form and back-transformation, i.e. the asymptotically important terms. By contrast, for n=2000 on a 64 processor Gamma, less than 10% of the time is spent on the asymptotically important terms. Latency, bandwidth and bisection all consume significant amounts of time. The $O(n)$ latency cost becomes less significant rapidly for fixed p as n increases. And, we have shown how to reduce the cost of bisection and inverse iteration on the Gamma by a factor of nearly 4. However, the bandwidth cost remains a problem.

The Basic Linear Algebra Communication Subroutines, `BLACS`, are designed to make it possible to greatly reduce the communication cost by coding architecture specific versions of the `BLACS`. At present the `BLACS` are built on top of vendor supplied communication libraries. If they were coded at a lower level, they could achieve substantial performance improvements at least for their collective communications routines. As shown by Karp et al.[10], collective communications can be performed in $n\tau_{band} + \tau_{lat} \lg p$ whereas at present they require $n\tau_{band} \lg p + \tau_{lat} \lg p$. Although this $\lg p$ factor appears small it is quite significant for the total running time of `PDSYEVX`.

Input and output data layout appears to be unimportant. Assuming that every processor owns roughly n^2/p elements of the input and output arrays, the cost of redistributing the data on input and/or on output is $O(n^2/p\tau_{band})$. This is significantly less than the $5n^2/\sqrt{p}\tau_{band}$ cost in reduction to tridiagonal form. Although we do not simulate different input and output data layouts, `PDSYEVX` performs an internal data redistribution which results in nearly every piece of an n^2 matrix being moved to another processor. This internal data redistribution never took more than 2.5% of the total time, which is strong evidence that redistributing most input or output layouts would also be relatively cheap.

Judging an algorithm by its implementation is dangerous. Our modeling led us to discover two simple performance improvements for bisection and inverse iteration. Initially, our model predicted much faster performance on the Intel Gamma than we actually obtained. This was traced to two factors. First, the default arithmetic on the Gamma includes a IEEE standard conforming divide operation which takes at least 50 times as long as multiply or add. We replaced this by a much faster, but less accurate divide. This requires a modification in the simple bisection algorithm to guarantee logical correctness despite possibly nonmonotonic arithmetic[5]. Second, a great deal of time was spent generating random numbers for inverse iteration. We changed from computing normally distributed random numbers, which require expensive transcendental function evaluations, to uniform random numbers. Together, these improvements sped up bisection and inverse iteration by a factor of nearly 4 on the Gamma.

References

[1] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A users' guide to PVM parallel virtual machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, Oak Ridge, TN, July 1991.

[2] C. Bischof, S. Huss-Lederman, X. Sun, A. Tsao, and T. Turnbull. Parallel performance of a symmetric eigensolver based on the invariant subspace decomposition approach. Technical report, Supercomputing Research Center, 1993. (Prism Working Note #15 ftp.super.org:pub/prism).

[3] J. Choi, J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Computer Society Press, 1992. (LAPACK Working Note #55).

[4] J. Choi, J. Dongarra, and D. Walker. *PB-BLAS: A set of Parallel Block Basic Linear Algebra Subprograms.* University of Tennessee, Knoxville, TN, 1993. available in postscript from netlib/scalapack.

[5] J. Demmel, I. Dhillon, and H. Ren. On the correctness of parallel bisection in floating point. Tech Report UCB//CSD-94-805, UC Berkeley Computer Science Division, March 1994. available via anonymous ftp from tr-ftp.cs.berkeley.edu, in directory pub/tech-reports/cs/csd-94-805, file all.ps.

[6] J. Demmel, M. Heath, and H. van der Vorst. Parallel numerical linear algebra. In A. Iserles, editor, *Acta Numerica, volume 2.* Cambridge University Press, 1993.

[7] F. Desprez, B. Tourancheau, and J. J. Dongarra. Performance complexity of *lu* factorization with efficient pipelining and overlap on a multiprocessor. Technical report, University of Tennessee, Knoxville, Feb 1994. (LAPACK Working Note #67).

[8] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[9] J. Dongarra, R. van de Geijn, and D. Walker. A look at scalable dense linear algebra libraries. In *Scalable High-Performance Computing Conference.* IEEE Computer Society Press, April 1992.

[10] R.M. Karp, A. Sahay, E. Santos, and K.E. Schauser. Optimal broadcast and summation in the LogP model. In *Proc. 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 142–153, 1993.

[11] B. Parlett. *The Symmetric Eigenvalue Problem.* Prentice Hall, Englewood Cliffs, NJ, 1980.

[12] J. Rutter. A serical implementation of Cuppen's divide and conquer algorithm for the symmetric eigenvalue problem. Mathematics Dept. Master's Thesis available by anonymous ftp to tr-ftp.cs.berkeley.edu, directory pub/tech-reports/cs/csd-94-799, file all.ps, University of California, 1994.

[13] R. Clint Whaley. Basic linear algebra communication subroutines: Analysis and implementation across multiple parallel architectures. Technical report, University of Tennessee, Knoxville, June 1994. (LAPACK Working Note #73).