Execution Time of Symmetric Eigensolvers

by

Kendall Swenson Stanley

B.S. (Purdue University) 1978

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor James Demmel, Chair
Professor William Kahan
Professor Phil Collela

Fall 1997

The dissertation of Kendall Swenson Stanley is approved:

Chair Date

Date

Date

University of California at Berkeley

Fall 1997

Execution Time of Symmetric Eigensolvers

by

Kendall Swenson Stanley

Abstract

Execution Time of Symmetric Eigensolvers

by

Kendall Swenson Stanley
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor James Demmel, Chair

The execution time of a symmetric eigendecomposition depends upon the application, the algorithm, the implementation, and the computer. Symmetric eigensolvers are used in a variety of applications, and the requirements of the eigensolver vary from application to application. Many different algorithms can be used to perform a symmetric eigendecompostion, each with differing computational properties. Different implementations of the same algorithm may also have greatly differing computational properties. The computer on which the eigensolver is run not only affects execution time but may favor certain algorithms and implementations over others.

This thesis explains the performance of the ScaLAPACK symmetric eigensolver, the algorithms that it uses, and other important algorithms for solving the symmetric eigenproblem on today's fastest computers. We offer advice on how to pick the best eigensolver for particular situations and propose a design for the next `ScaLAPACK` symmetric eigensolver which will offer greater flexibility and 50% better performance.

Professor James Demmel
Dissertation Committee Chair

To the memory of my father. My most ambitious goal is to be as good a father as he was to me.

Contents

List of Figures

List of Tables

Acknowledgements

I thank those that I have worked with during my wonderful years at Berkeley. Doug Ghormley taught me all that I know about emacs, X, and tcsh. Susan Blackford, Clint Whaley and Antoine Petitet patiently answered my stupid questions about ScaLAPACK. I thank Bruce Hendrickson for numerous insights. Mark Sears and Greg Henry gave me the opportunity to test out some of my ideas on a real application. Peter Strazdins' study of software overhead convinced me to take a hard look at code cache misses. Ross Moore gave me numerous typesetting hints and suggestions. Beresford Parlett helped me with the section on Jacobi. Oliver Sharp helped convince me to ask Jim Demmel to be my advisor and gave some early help with technical writing. I am indebted to the members of the ScaLAPACK team whose effort made ScaLAPACK, and hence this thesis, possible.

My graduate studies would not have been possible were it not for my friends and family who encouraged me to resume my education and continued to support me in that decision, especially my wife (Marta Laskowski), Greg Lee, and Marta's parents Michael and Joan. I also thank Chris Ranken for his friendship; my parents for bringing me into a loving world and teaching me to love mathematics; and Howard and Nani Ranken who proved, by example, that the two boby problem can be solved and inspired Marta and I to pursue the dream of two academic careers in one household.

I thank the members of my committee for their help and advice. I thank my advisor for allowing me the luxury of doing research without worrying about funding[1] or machine access at UC Berkeley[2] and the University of Tennesse at Knoxville[3] I thank Prof. Kahan for his sage advice, not just on the technical aspects, but also on the non-technical aspects of a research career and on life itself. I thank Phil Colella for his interest in my work and for reading my thesis on extremely short notice.

Most importantly, I thank my wife for her love and never ending support and I thank my daughter for making me smile.

[1]This work was supported primarily by the Defense Advanced Research Projects Agency of the Department of Defense under contracts DAAL03-91-C-0047 and DAAH04-95-1-0077, and with additional support provided by the Department of Energy grant DE-FG03-94ER25206. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

[2]National Science Foundation Infrastructure grant Nos. CDA-9401156 and CDA-8722788.

[3]The University of Tennessee, Knoxville, acquired the IBM SP2 through an IBM Shared University Research Grant. Access to the machine and technical support was provided by the University of Tennessee/Oak Ridge National Laboratory Joint Institute for Computational Science.

Part I

First Part

Chapter 1

Summary - Interesting Observations

The symmetric eigendecomposition of a real symmetric matrix is: $A = QDQ^T$, where D is diagonal and Q, is orthonormal, i.e. $Q^TQ = I$. Tridiagonal based methods reduce A to a tridiagonal matrix through an orthonormal similarity transformation, i.e. $A = ZTZ^T$, compute the eigendecomposition of the tridiagonal matrix $T = UDU^T$ and, if necessary, transform the eigenvectors of the tridiagonal matrix back into eigenvectors of the original matrix A, i.e. $Q = ZU$. Non-tridiagonal based methods operate directly on the original matrix A.

I am interested in understanding and minimizing the execution time of dense symmetric eigensolvers, as used in real applications, on distributed memory parallel computers. I have modeled the performance of symmetric eigensolvers as a function of the algorithm, the application, the implementation and the computer. Some applications require only a partial eigendecomposition, i.e. only a few eigenvalues or eigenvectors. Different implementations may require different communication or computation patterns and they may use different libraries and/or compilers. This thesis concentrates on the $O(n^3)$ cost of reduction to tridiagonal form and transforming the eigenvectors back to the original space.

I have modeled the execution time of the `ScaLAPACK`[31] symmetric eigensolver, `PDSYEVX`, in detail and validated this model against actual performance on a number of distributed memory parallel computers. `PDSYEVX`, like most `ScaLAPACK` codes, uses calls to the `PBLAS`[41, 140] to perform basic linear algebra operations such as matrix-matrix multiply and matrix-vector multiply in parallel. PDSYEVX and the PBLAS use calls to the Basic Linear Algebra Subroutines, BLAS[63, 62], to perform basic linear algebra operations such as matrix-matrix multiply and matrix-vector multiply on data local to each processor, and calls to the Basic Linear Algebra Communications Subroutines, BLACS[169, 69], to move data between the processors. The level one BLAS involving only vectors and perform $O(n)$ flops on $O(n)$ data, where n is the length of the vector. The level two BLAS involve one matrix and one or two vectors and perform $O(n^2)$ flops on $O(n^2)$ data, where the matrix is of size $n \times n$. The level three BLAS involve only matrices and perform $O(n^3)$ flops on $O(n^2)$ data and offer the best opportunities to obtain peak floating point performance through data re-use.

PDSYEVX uses a 2D block cyclic data layout for all input, output and internal matrices. 2D block cyclic data layouts have been shown to support scalable high performance parallel dense linear algebra codes[32, 30, 124] and hence have been selected as the primary data layout for HPF[110], ScaLAPACK[68] and other parallel dense linear algebra libraries[98, 164]. A 2D block cyclic data layout is defined by the processor grid (p_r by p_c), the local block size (mb by nb) and the location of the $(1,1)$ element of the matrix. In this thesis, we will assume that the $(1,1)$ element of matrix A, i.e. $A(1,1)$ is mapped to the $(1,1)$ element of the local matrix in processor $(0,0)$. Hence, $A(i,j)$ is stored in element $(\lfloor \frac{i-1}{\mathsf{mb} \times p_r} \rfloor \mathsf{mb} + \mod(i-1, \mathsf{mb} \times p_r) + 1, \lfloor \frac{j-1}{\mathsf{nb} \times p_c} \rfloor \mathsf{nb} + \mod(j-1, \mathsf{nb} \times p_c) + 1)$ on processor $(\mod(\lfloor \frac{i-1}{\mathsf{mb}} \rfloor, p_r), \mod(\lfloor \frac{j-1}{\mathsf{nb}} \rfloor, p_c))$. Figures 1.1 and 1.2, reprinted from the ScaLAPACK User's Guide[31] shows how a 9 by 9 matrix would be distributed over a 2 by 3 processor grid with $\mathsf{mb} = \mathsf{nb} = 2$. In general, we will assume that square blocks are used since this is best for the symmetric eigenproblem, and we will use nb to refer to both the row block size and the column block size.

All ScaLAPACK codes including PDSYEVX in version 1.5 use the data layout block size as the algorithmic blocking factor. Hence, except as noted, we use nb to refer to the algorithmic blocking factor as well as the data layout block size. Data layouts, and algorithmic blocking factors are discussed in Section 2.3.3.

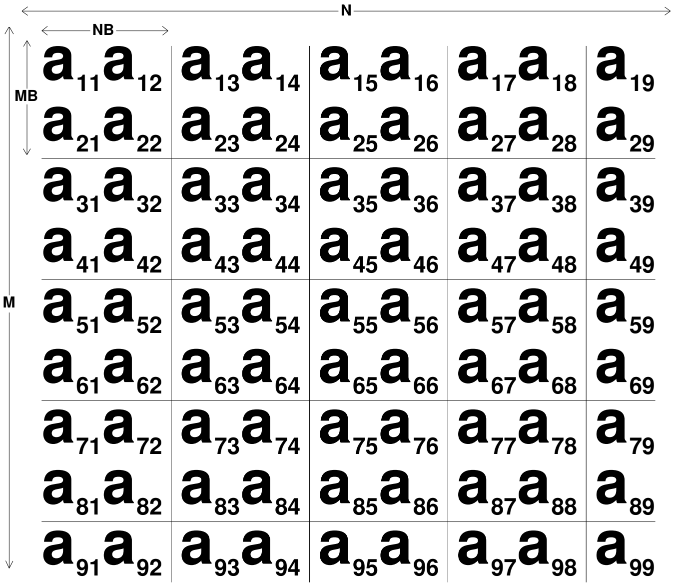
PDSYEVX calls the following routines:

PDSYTRD Performs Householder reduction to tridiagonal form.

PDSTEBZ Computes the eigenvalues of a tridiagonal matrix using bisection.

PDSTEIN Computes the eigenvectors of the tridiagonal matrix using inverse iteration and

Figure 1.1: 9 by 9 matrix distributed over a 2 by 3 processor grid with $\mathsf{mb} = \mathsf{nb} = 2$



9 x 9 matrix partitioned in 2 x 2 blocks

Gram-Schmidt reorthogonalization.

`PDORMTR` Transforms the eigenvectors of the tridiagonal matrix back into eigenvectors of the original matrix.

My performance models explain performance in terms of the following application parameters:

n The matrix size.

m The number of eigenvectors required.

e The number of eigenvalues required. ($e \geq m$)

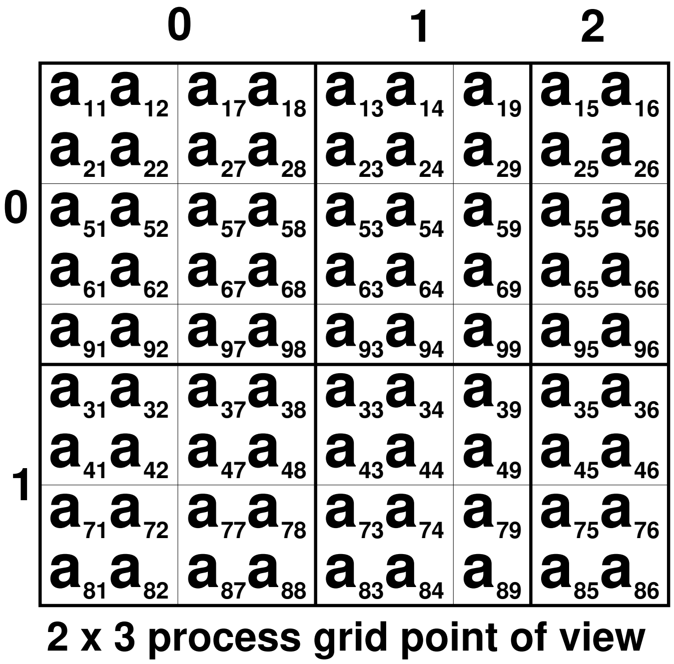
the following machine parameters:

p The number of processors (arranged in a p_r by p_c grid as described below).

α The communication latency (secs/message).

β The inverse communication bandwidth (secs/double precision word). This means that sending a message of k double precision words costs: $\alpha + k\beta$.

Figure 1.2: Processor point of view for 9 by 9 matrix distributed over a 2 by 3 processor grid with mb = nb = 2



$\gamma_1, \gamma_2, \gamma_3$ Time per flop for BLAS1, BLAS2 and BLAS3 routines respectively.

$\delta_1, \delta_2, \delta_3, \delta_4$ Software overhead for BLAS1, BLAS2, BLAS3 and PBLAS routines respectively. This means that a call to DGEMM(a BLAS3 routine) requiring c flops costs: $\delta_3 + c\gamma_3$. See Chapter 3 for details on the cost of the BLAS. The cost of the PBLAS routine PDSYMV is shown in Table 4.3.

My model also uses the following algorithmic and data layout parameters:

p_r The number of processor rows in the processor grid.

p_c The number of processor columns in the processor grid.

nb The data layout block size and algorithmic blocking factor.

These and all other variables used in this thesis are listed in Table A.1 in Appendix A.

The rest of this chapter presents the most interesting results from my study of the execution time of symmetric eigensolvers on distributed memory computers. Section 1.1

describes the algorithms commonly used for dense symmetric eigendecomposition on distributed memory parallel computers. Section 1.2 describes how software overhead and load imbalance costs are significant. Section 1.3 explains the two rules of thumb for ensuring that a distributed memory parallel computer can achieve good performance on a dense linear algebra code such as `ScaLAPACK`'s symmetric eigensolver. Section 1.4 explains that it is important to identify which techniques offer the greatest potential for improving performance across a wide range of applications, computers, problem sizes and distributed memory parallel computers. Section 1.5 gives a synopsis of how execution time of the `ScaLAPACK` symmetric eigensolver could be reduced. Section 1.6 explains the types of applications on which Jacobi can be expected to be as fast as, or faster than, tridiagonal based methods.

The rest of my thesis is organized as follows. Chapter 2 provides an introduction and a historical prospective. Chapter 3 explains the performance of the Basic Linear Algebra Subroutines (BLAS). Chapter 4 contains my complete execution time model for `ScaLAPACK`'s symmetric eigensolver, `PDSYEVX`. Chapter 5 simplifies the execution time model by concentrating on a particular application on a particular distributed memory parallel computer, the Intel Paragon. Chapter 6 explains the performance requirements of distributed memory parallel computers and discusses the execution time of `PDSYEVX`. Chapter 7 explains the performance of other dense symmetric eigensolvers. Chapter 8 provides a blueprint for reducing the execution time of `PDSYEVX`. Chapter 9 offers concise advice to users of symmetric eigensolvers.

1.1 Algorithms

There are many widely disparate symmetric eigendecomposition algorithms. Tridiagonal reduction based algorithms for the symmetric eigendecomposition require asymptotically the fewest flops and have been historically the fastest and most popular[83, 79, 129, 153, 86, 145, 134, 50].

Iterative eigensolvers, e.g. Lanczos and conjugate gradient methods, are clearly superior if the input matrix is sparse and only a limited portion of the spectrum is needed[49, 119]. Iterative eigensolvers are out of the scope of this thesis.

Even for tridiagonal matrices, there are several algorithms worthy of attention for the tridiagonal eigendecomposition. The ideal method would require at most $O(n^2)$ floating point operations, $O(n)$ message volume and $O(p)$ messages. The recent work of

Parlett and Dhillon[136, 139] renews hope that such a method will be available in the near future. Should this effort hit unexpected snags, other better known methods, such as QR[79, 86, 93], QD[135], bisection and inverse iteration[83, 102] and Cuppen's divide and conquer algorithm[50, 66, 147, 88] will remain common. Parallel codes have been written for QR[39, 8, 76, 125], bisection and inverse iteration[15, 75, 54, 81] and Cuppen's algorithm[82, 80, 141]. `ScaLAPACK` offers parallel QR and parallel bisection and inverse iteration codes and Cuppen's algorithm[50, 66, 88], which has recently replaced QR as the fastest serial method[147], has been coded for inclusion in `ScaLAPACK` by Françoise Tisseur. Algorithms for the tridiagonal eigenproblem are discussed in Section 2.2, and parallel tridiagonal eigensolvers are discussed in Section 7.1.

A detailed comparison of tridiagonal eigensolvers would be premature until Parlett and Dhillon complete their prototype.

This thesis concentrates on the $O(n^3)$ cost of reduction to tridiagonal form and transforming the eigenvectors back to the original space. Hendrickson, Jessup and Smith[91] showed that reduction to tridiagonal form can be performed 50% faster than `ScaLAPACK` does. Lang's successive band reduction[116], SBR, is interesting at least if only eigenvalues are to be computed. But the complexity of SBR has made it difficult to realize the theoretical advantages of SBR in practice. A performance model for `PDSYEVX`, `ScaLAPACK`'s symmetric eigensolver, section 7.1.2. is given in Chapter 4. By restricting our attention to a single computer, and to the most common applications, the model is further simplified and discused in Chapter 5.

Jacobi requires 4-20 times as many floating point operations as tridiagonal based methods, hence the type of problems on which Jacobi will be faster will always be limited. Jacobi is faster than tridiagonal based methods[125, 2] on small spectrally diagonally dominant matrices[1] despite requiring 4 times as many flops because it has less overhead. However, on large problems tridiagonal based methods can achieve at least 25% efficiency and will hence be faster than any method requiring 4 times as many flops. And, on matrices that are not spectrally diagonally dominant, Jacobi requires 20 or more times as many flops as tridiagonal based methods - a handicap that is simply too large to overcome. Jacobi's method is discussed in Section 7.3.

Methods that require multiple n by n matrix-matrix multiplies, such as the Invari-

[1] Spectrally diagonally dominant means that the eigenvector matrix, or a permutation thereof, is diagonally dominant.

ant Subspace Decomposition Approach[97] (ISDA), and Yau and Lu's FFT based method[174] require roughly 30 times as many floating point operations as tridiagonal based methods and hence may never be faster than tridiagonal based methods. The ISDA for solving symmetric eigenproblems is discussed in Section 7.4.

Banded ISDA[26] is an improvement on ISDA that involves

an initial bandwdith reduction. Banded ISDA[26] is nearly a tridiagonal method and offers performance that is nearly as good, at least if only eigenvalues are sought. However since a banded ISDA code requires multiple bandwidth reduction each of which requires a back transformation, if even a few eigenvectors are required, a banded ISDA code must either store the back transformations in compact form or it will perform an additional $O(n^3)$ flops. No code available today stores and applies these backtransformations in compact form. At present, the fastest banded ISDA code starts by reducing the matrix to tridiagonal form and is neither the fastest tridiagonal eigensolver, nor the easiest to parallelize. Banded ISDA is discussed in Section 7.5.

In conclusion, reduction to tridiagonal form combined with Parlett and Dhillon's tridiagonal eigensolver is likely to be the preferred method for eigensolution of dense matrices for most applications.

In the meantime, until Parlett and Dhillon's code is available, we believe that `PDSYEVX` is the best general purpose symmetric eigensolver for dense matrices. It is available on any machine to which `ScaLAPACK` has been ported[2], it achieves 50% efficiency even when the flops in the tridiagonal eigensolution are not counted[3] and it scales well, running efficiently on machines with thousands of nodes. It is faster than ISDA and faster than Jacobi on large matrices and on matrices that are not spectrally diagonally dominant.

1.2 Software overhead and load imbalance costs are significant

In `PDSYEVX`, it is somewhat surprsing but true that software overhead and load imbalance costs are larger than communications costs. In its broadest definition, software overhead is the difference between the actual execution time and the cost of communication

[2]Intel Paragon, `Cray T3D`, `Cray T3E`, `IBM SP2`, and any machine supporting the `BLACS`, `MPI` or `PVM`

[3]Our definition of efficiency is a demanding one: total time divided by the time required by reduction to tridiagonal form and back transformation assuming that these are performed at the peak floating point execution rate of the machine. i.e. $time/(\frac{10}{3}\frac{n^3}{p}\gamma_3)$

and computation. Software overhead includes saving and restoring registers, parameter passing, error and special case checking as well as those tasks which prevent calls to the BLAS involving few flops from being as efficient as calls to the BLAS involving many flops: loop overhead, border cases and data movement between memory hierarchies that gets amortized over all the operations in a given call to the BLAS. The cost of any operation which is performed by only a few of the processors (while the other processors are idle) is a load imbalance cost.

Because software overhead is as significant as communication latency, the three term performance model introduced by Choi et al.[40] and used in my earlier work[57], which only counts flops, number of messages and words communicated, does not adequately model the performance of PDSYEVX. In addition to these three terms a fourth term, which we designate δ, representing software overhead costs is required.

Software overhead is more difficult to measure, study, model and reason about than the other components of execution time. Measuring the execution time required for a subroutine call requiring little or no work measures only subroutine call overhead, parameter passing and error checking. For the performance models in this thesis, we measure the execution time of each routine across a range of problem sizes (with code cached and data not cached) and use curve fitting to estimate the software overhead of an individual routine. Because we perform these timings with code cached but data not cached, this gives an estimate of all software overhead costs except code cache misses.

We use times with the code cached and data for our performance models because, for most problem sizes, the matrix is too large to fit in cache but it is less clear whether code fits in cache or not. It is easy to compute the amount of data which must be cached, but there is no portable automatic way to measure the amount of code which must be cached. Furthermore, the data cache needs, for typical problem sizes, are much larger than code cache needs, hence while it is usually clear that the data is not cached the code cache needs and code cache size are much closer.

A full study of software overhead costs is out of the scope of this thesis and remains a topic for future research. The overhead and load imbalance terms in the performance model for PDSYEVX on the PARAGON are explained in Sections 5.7 and 5.8.

1.3 Effect of machine performance characteristics on PDSYEVX

The most important machine performance characteristic is the peak floating point rate. Bisection bandwidth essentially defines which machines ScaLAPACK can perform well on. Message latency and software overhead, since they are $O(n)$ terms are important primarily for small and medium matrices.

Most collections of computers fall into one of two groups: those connected by a switched network whose bisection bandwdith increases linearly (or nearly so) with the number of processors and those connected by a network that only allows one processor to send at a time. All current distributed memory parallel computers that I am aware of have adequate bisection bandwdith[4] to support good efficiency on PDSYEVX. On the other hand, no network that only allows one processor to send at a time can allow scalable performance and none that I am aware of allows good performance with as many as 16 processors. As long as the bandwidth rule of thumb (explained in detail in Section 6.1.1) holds, bandwidth will not be the limiting factor in the performance of PDSYEVX.

Bandwidth rule of thumb: Bisection bandwidth per processor[5] *times the square root of memory size per processor should exceed floating point performance per processor.*

$$\frac{\text{Megabytes/sec}}{\text{processor}} \times \frac{\sqrt{\text{Megabytes}}}{\text{processor}} > \frac{\text{Megaflops/sec}}{\text{processor}}$$

assures that bandwidth will not limit performance.

Assuming that the bandwidth is adequate, we consdier next the problem size per processor:

If the problem is large enough, i.e. $(n^2/p) > 2 \times (Megaflops/processor)$, then PDSYEVX should execute reasonably efficiently. This rule (explained in detail in Section 6.1.2 can be restated as:

Memory size rule of thumb: memory size should match floating point performance

Memory size rule of thumb: memory size should match floating point performance

[4] Few distributed memory parallel computers offer bandwidth that scales linearly with the number of processors but most still have adequate bisection bandwidth.

[5] Bisection bandwidth per processor is the total bisection bandwidth of the network divided by the number of processors.

$$\frac{\text{Megabytes}}{\text{processor}} > \frac{\text{Megaflops/sec}}{\text{processor}}$$

assures that `PDSYEVX` will be efficient on large problems.

If the problem is not large enough, lower order terms, as explained in Chapter 4 will be significant. Unlike the peak flop rate which can be substantially independent of main memory performance, lower order terms (communication latency, communication bandwidth, software overhad and load imbalance) are strongly linked to main memory performance.

`PDSYEVX` can work well on machines with large slow main memory (on large problems) and or machines with small fast main memory (on small problems). Most distributed memory parallel computers have sufficient memory size and network bisection bandwidth to allow `PDSYEVX` to achieve high efficiency on large problem sizes. The `Cray T3E` is one of the few machines that has sufficient main memory performance to allow `PDSYEVX` to achieve high performance on small problem sizes. The effect of machine performance characteristics on `PDSYEVX` is discussed in Chapter 6.

1.4 Prioritizing techniques for improving performance.

One fo the most importatn uses of performance modeling is to identify which techniques offer the most promise for performance improvement, because there are too many performance improvement techniques to allow one to try them all. One technique that appeared to be important early in my work, optimizing global communications, now appears less important in light of the discovery that software overhead and load imbalance are more significant than earlier thought. Here we talk about general conclusions; details are summarized in Section 1.5, and elaborated in Chapters 7 and 8.

Overlapping communication and computation, though it undeniably increases performance, should be implemented only after every effort has been made to reduce both communications and computations costs as much as possible. Overlapping communication and computation has proven to be more attractive in theory than in practice because not all communication costs overlap well and communication costs are not the only impediment to good parallel performance.

Although Strassen's matrix multiplication has been proven to offer performance better than can be achieved through traditional methods, it will be a long time before a Strassen's matrix multiply is shown to be twice as fast as a traditional method. A typical single processor computer would require 2-4 Gigabytes of main memory to achieve an effective flop rate of twice the machine's peak flop rate[6] and 2-4 Terabytes of main memory to achieve 4 times the peak flop rate. Strassen's matrix multiplication will get increasing use in the coming years, because achieving 20% above "peak" performance is nothing to sneeze at, but Strassen's matrix multiply will not soon make matrix multiply based eigendecomposition such as ISDA faster than tridiagonal based eigendecomposition.

1.5 Reducing the execution time of symmetric eigensolvers

`PDSYEVX` can be improved. It does not work well on matrices with large clusters of eigenvalues. And, it is not as efficient as it could be[91], achieving only 50% of peak efficiency on `PARAGON`, `Cray T3D` and `Berkeley NOW` even on large matrices. On small matrices it performs worse. Parlett and Dhillon's new tridiagonal eigensolver promises to solve the clustered eigenvalue problem so we concentrate on improving the performance of reduction to tridiagonal form and back transformation.

Input and output data layout need not affect execution time of a parallel symmetric eigensolver because data redistribution is cheap. Data redistribution requires only $O(p)$ messages and $O(n^2/p)$ message volume per processor. This is modest compared to $O(n \log(p))$ messages and $O(n^2/\sqrt{p})$ message volume per processor required by reduction to tridiagonal form and back transformation.

Separating internal and external data layout actually decreases minimum execution time over all data layouts. Separating internal and external data layouts allows reduction to tridiagonal form and back transformation to use different data layouts. It also allows codes to concentrate only on the best data layout, reducing software overhead and allowing improvements which would be prohibitively complicated to implement if they had to work on all two-dimensional block cyclic data layouts.

Separating internal and external data layouts increases the minimum workspace requirement[7] from $2.5\,n^2$ to $3\,n^2$. However with minor improvements in the existing code,

[6] A dual processor computer would require twice as much memory.

[7] Assuming that data redistribution is not performed in place. It is difficult to redistribute data in place

and without any changes to the interface, internal and external data layout can be separated without increasing the workspace requirement. See Section 8.5.

Lichtenstein and Johnson[124] point out that data layout is irrelevant to many linear algebra problems because one can solve a permuted problem instead of the original. This works for symmetric problems provided that the input data is distributed over a square processor grid and with a row block size is equal to the column block size.

Hendrickson, Jessup and Smith[91] demonstrated that the performance of `PDSYEVX` can be improved substantially by reducing load imbalance, software overhead and communications costs. Most of the inefficiency in `PDSYEVX` is in reduction to tridiagonal form. Software overhead and load imbalance are responsible for more of the inefficiency than the cost of communications. Hence, it is those areas that need to be sped up the most. Preliminary results[91] indicate that by abandoning the `PBLAS` interface, using `BLAS` and `BLACS` calls directly, and concentrating on the most efficient data layout, software overhead, load imbalance and communications costs can be cut in half. Strazdins has investigated reducing software overheads in the `PBLAS`[161], but it remains to be seen whether software overheads in the `PBLAS` can be reduced sufficiently to allow `PDSYEVX` to be as efficient as it could be. `PDSYEVX` performance can be improved further if the compiler can produce efficient code on simple doubly nested loops, implementing merged `BLAS` Level 2 operations (like `DSYMV` and `dsyr2`).

For small matrices, software overhead dominates all costs, and hence one should minimize software overhead even at the expense of increasing the cost per flop. An unblocked code has the potential to do just that.

Although back transformation is more efficient than reduction to tridiagonal form, it can be improved. Whereas software overhead is the largest source of inefficiency in reduction to tridiagonal form, communications cost and load imbalance are the largest source of inefficiency in back transformation. Load imbalance is hard to eliminate in a blocked data layout in reduction to tridiagonal form because the size of the matrix being updated is constantly changing (getting smaller), but in back transformation, all eigenvectors are constantly updated, so statically balancing the number of eigenvalues assigned to each processor works well. Therefore the best data layout for back transformation is a two-dimensional rectangular block-cyclic data layout. The number of processor columns, p_c,

between two arbitrary parallel data layouts. If efficient in-place data redistribution were feasible, separating internal and external data layout would require only a trivial increase in workspace.

should exceed the number of processor rows by a factor of approximately 8. The optimal data layout column block size is: $\lceil n/(p_c k) \rceil$ for some small integer k. The row *blocksize* is less important in back transformation, and 32 is a reasonable choice, although setting it to the same value as the column block size will also work well if the `BLAS` are efficient on that block size and $p_r < p_c$. Many techniques used to improve performance in LU decomposition, such as overlapping communication and computation, pipelining communication and asynchronous message passing can also be used to improve the performance of back transformation. Of these techniques, only asynchronous message passing (which eliminates all local memory movement) requires modification to the `BLACS` interface. The modification to the `BLACS` needed to support asynchronous message passing would allow forward and backward compatibility.

All of these methods are discussed in Chapter 8.

1.6 Jacobi

A one-sided Jacobi method with a two-dimensional data layout will beat tridiagonal based eigensolvers on small spectrally diagonally dominant matrices. The simpler one-dimensional data layout is sufficient for modest numbers of processors, perhaps as many as a few hundred, but does not scale well. Tridiagonal based methods, because they require fewer flops, will beat Jacobi methods on random matrices regardless of their size on large ($n > 200\sqrt{p}$) matrices even if they are spectrally diagonally dominant. Jacobi also remains of interest in some cases when high accuracy is desired[58]. Jacobi's method is discussed in Section 7.3.

1.7 Where to obtain this thesis

This thesis is available at: `http://www.cs.berkeley.edu/stanley/thesis`

Chapter 2

Overview of the design space

2.1 Motivation

The execution time of any computational solution to a problem is a single-valued function (time) on a multi-dimensional and non-uniform domain. This domain includes the problem being solved, the algorithm, the implementation of the algorithm and the underlying hardware and software (sometimes referred to collectively as the computer). By studying one problem, the symmetric eigenproblem, in detail we gain insight into how each of these factors affects execution time.

Section 2.2 discusses the most important algorithms for dense symmetric eigendecomposition on distributed memory parallel computers. Section 2.3 discusses the effect that the implementation can have on execution time. Section 2.4 discusses the effect of various hardware characteristics on execution time. Section 2.5 lists several applications that uses symmetric eigendecomposition and their differing needs. Section 2.6 discusses the direct and indirect effects of machine load on the execution time of a parallel code. Section 2.7 outlines the most important historical developments in parallel symmetric eigendecomposition.

2.2 Algorithms

The most common symmetric eigensolvers which compute the entire eigendecomposition use Householder reduction to tridiagonal form, form the eigendecomposition of the tridiagonal matrix and transform the eigenvectors back to the original basis. Algorithms that do not begin by reduction to tridiagonal form require more floating point operations.

Except for small spectrally diagonally dominant matrices, on which Jacobi will likely be faster than tridiagonal based methods, and scaled diagonally dominant matrices on which Jacobi is more accurate[58], tridiagonal based codes will be best for the eigensolution of dense symmetric matrices. See Section 7.3 for details.

The recent work of Parlett and Dhillon offers the promise of computing the tridiagonal eigendecomposition with $O(n^2)$ flops and $O(p)$ messages. Should some unexpected hitch prevent this from being satisfactory on some matrix types, there are several other algorithms from which to choose. Experience with existing implementations shows that for most matrices of size 2000 by 2000 or larger, the tridiagonal eigendecomposition is a modest component of total execution time.

Reduction to tridiagonal form and back transformation are the most time consuming steps in the symmetric eigendecomposition of dense matrices. These two steps require more flops ($O(n^3)$ vs. $O(n^2)$), more message volume ($O(n^2\sqrt{p})$ vs. $O(n^2)$) and more messages ($O(n\log(p))$ vs. $O(p)$) than the eigendecomposition of the tridiagonal matrix. Since the cost of the matrix transformations (reduction to tridiagonal form and back transformation) grows faster than the cost of tridiagonal eigendecomposition, the matrix transformations are the dominant cost for larger matrices.

Reduction to tridiagonal form and back transformation require different communication patterns. Reduction to tridiagonal form is a two-sided transformation requiring multiplication by Householder reflectors from both the left and right side. Two sided reductions require that every element in the trailing matrix be read for each column eliminated, hence half of the flops are `BLAS2` matrix-vector flops and $O(n\log(p))$ messages are required.

Equally importantly, two-sided reductions require significant calculations within the inner loop, which translates into large software overhead. Indeed on the computers that we considered, software overhead appears to be a larger factor in limiting efficiency of reduction to tridiagonal form than communication.

Back transformation is a one-sided transformation with updates than can be formed anytime prior to their application. Hence back transformation requires $O(n/\mathtt{nb})$ messages (where `nb` is the data layout block size) and far less software overhead than reduction to tridiagonal form.

Chapters 4 and 5 discuss the execution time of reduction to tridiagonal form and back transformation, as implemented in `ScaLAPACK`, in detail.

2.3 Implementations

2.3.1 Parallel abstraction and languages

There are three common ways of expressing parallelism in linear algebra codes: message passing, shared memory and calls to the BLAS. Message passing programs tend to keep communication to a minimum, in part because the communication is specified directly. Shared memory codes can outperform message passing codes when load imbalance costs outweigh communication costs[118]. All calls to the BLAS offer potential parallelism though the potential for speedup varies. ScaLAPACK uses message passing while LAPACK exposes parallelism through calls to the BLAS.

In some cases, recent compilers are able to identify the parallelism in codes that may not have been written specifically for parallel execution[172, 171]. However, experience has shown that programs designed for sequential machines rarely exhibit the properties necessary for efficient parallel execution, hence some research into parallelizing compilers has switched its emphasis to parallelizing codes which are written in languages such as HPF[94, 110] which allow the programmer to express parallelism and allow some control over data layout.

Codes written in any standard sequential language, such as C, C++ or Fortran can achieve high performance, especially if the majority of the operations are performed within calls to the BLAS. If the flops are performed within codes written in the language itself, the execution time will depend upon the code and the compiler more than on the language used. If pointers are used carelessly in C, the compiler may not be able to determine the data dependencies exactly and may have to forgo certain optimizations[172]. On the other hand, carefully crafted C codes, tuned for individual architectures and compiled with modern optimizing compilers can result in performance that rivals that of carefully tuned assembly codes[23, 168].

2.3.2 Algorithmic blocking

A blocked code is one that has been recast to allow some of the flops to be performed as efficient BLAS3 matrix-matrix multiply flops[6, 4]. Typically a block of columns is reduced using an unblocked code followed by a matrix-matrix update of the trailing matrix. The algorithmic blocking factor is the number of columns (or rows) in the block column.

In serial codes, data layout blocking does not exist and hence the algorithmic blocking factor is referred to simply as the blocking factor. In ScaLAPACK version 1.5, the algorithmic blocking factor is set to match the data layout blocking factor.

2.3.3 Internal Data Layout

Most of the flops in blocked dense linear codes involve a rank k update, i.e. $A' = A + B * C$ where $A \in R^{m,n}$, $B \in R^{m,k}$, $C \in R^{k,n}$ and m, n are $O(n)$ and k is the algorithmic blocking factor (a tuning parameter typically much smaller than n or m). A may be triangular and B and/or C may be transposed or conjugate transposed. Hence internal data layout must support good performance on such rank k updates.

A is typically updated in place, i.e. the node which owns element $A_{i,j}$ computes and stores $A'_{i,j}$. This is called the owner computes rule and is motivated by the high cost of data movement relative to the cost of floating point computation. If k is large enough a 3D data layout is more efficient[1] [12], and performance can be improved further by using Strassen's matrix multiply[157] [96] [70]. Some dense linear algebra codes, including LU, can be recursively partitioned[165] resulting in large values of k for the majority of the flops. Nonetheless, though a 3D data layout might be best for a recursively partitioned LU, reduction to tridiagonal form is most efficient with a modest algorithmic blocking factor and hence it is more efficient to update A in place and we will make that assumption for the rest of this discussion.

If A is to be updated in place, a 2D layout minimizes the total communication requirement for rank k updates. The elements of B and C which must be sent to each node are determined by the elements of A owned by that node. The node that owns element $A_{i,j}$ must obtain a copy of row i of B and column j of C. The number of elements of matrices B and C that a given node must obtain is k times the number of rows and columns of A for which the node owns at least one element. If a node must own r^2 elements, the number of elements of B and C which must be obtained is minimized if the node owns a square submatrix of A corresponding to r rows and r columns. In a 2D layout, the processors are arranged in a rectangular grid. Each row of the matrix is assigned to a row of the processor grid. Each column is assigned to a column of the processor grid.

The common ways of assigning the rows and columns to the processor grid in a 2D layout are: block, cyclic and block-cyclic. For the following descriptions, we will assume that we are distributing n rows of A over p_r processor rows. In a cyclic layout, row i is assigned to processor row $i \bmod p_r$. In a block layout, row i is assigned to processor row $\lfloor \frac{i-1}{\lceil n/p_r \rceil} \rfloor$. In a block-cyclic data layout, row i is assigned to processor row $\lfloor \frac{i-i}{\mathsf{nb}} \bmod p_r \rfloor$, where nb is the data layout block-size. The block-cyclic data layout includes the other two as special cases.

Block-cyclic data layouts simplify algorithmic blocking and are used in most parallel dense linear algebra libraries[68] [98, 164]. However, by separating algorithmic blocking from data blocking it is usually[1] possible to achieve high performance from a cyclic data layout[91, 140, 44, 158].

One-dimensional data layouts require $O(n^2)$ data movement per node (compared to $O(n^2/\sqrt{p})$ for 2D data layouts) and are generally less efficient. However, there are certain situations in which 1D data layouts are preferred. If the communication pattern is strictly one-dimensional (i.e. only along rows or columns) a 1D data layout requires no communication. Furthermore, some applications, such as LU, require much more communication in one direction than the other[2]. Hence, for modest numbers of processors it may be better to use a 1D data layout.

A square processor grid can greatly simplify symmetric reductions - allowing lower overhead codes. Furthermore, I believe that pipelining and lookahead (see section 2.4.2) can only be used effectively on symmetric reductions (such as Cholesky and reduction from generalized to standard form) when a square processor grid is used[3].

All existing parallel dense linear algebra libraries use the same input data layout as the internal data layout. In Chapter 8 I will demonstrate that this is not necessary to achieve high performance and that in fact performance can be improved by using a different data layout internally than the input and output data layout.

[1] Block-cyclic data layouts still maintain an advantage over cyclic data layouts on machines with high communication latency, especially in those algorithms, such as Cholesky and back transformation, that require only $O(n/\mathsf{nb})$ messages, where nb is the data layout block-size.

[2] LU with partial pivoting requires $O(n \log(p))$ messages within the processor columns but only $O(n/\mathsf{nb})$ messages within the processor rows[31, 40, 30]. The total volume of communication however is similar in both directions.

[3] Pipelining and lookahead cannot be used in reduction to tridiagonal form because of its synchronous nature.

2.3.4 Libraries

Software libraries can improve portability, robustness, performance and software re-use. ScaLAPACK is built on top of the BLAS and BLACS and hence will run on any system on which a copy of the BLAS[63, 62] and BLACS[169, 69] can be obtained.

Libraries, and their interface, have both a positive and a negative effect on performance. The existence of a standard interface to the BLAS means that by improving the performance of a limited set of routines, i.e. the BLAS, one can improve the performance of the entire LAPACK and ScaLAPACK library and other codes as well. Hence, many manufacturers have written optimized BLAS for their machines. In addition, Bilmes et al.[23, 168] have written a portable high performance matrix-matrix multiply and two other research groups have written high performance BLAS that depend only on the existence of a high performance matrix-matrix multiply[51, 103, 104]. Portable high performance BLAS offers the promise of high performance on LAPACK and ScaLAPACK codes without the expense of hand coded BLAS.

However, adhering to a particular library interface necessarily rules out some possibilities. The BLACS do not support asynchronous receives, a costly limitation on the Paragon. The BLAS do not meet all computational needs[108], especially in parallel codes[91], hence the programmer is faced with the choice of reformulating code to use what the BLAS offers or avoiding the BLAS and trusting the compiler to produce high performance code. Furthermore, the interface itself implies some overhead, at the very least a subroutine call but typically much more than that[161]. Strazdins[161] showed that software overhead in ScaLAPACK accounts for 15-20% of total execution time even for the largest problems that fit in memory on a Fujitsu VP1000.

2.3.5 Compilers

Compiler code generation is relatively unimportant to LAPACK and ScaLAPACK performance, because these codes are written so that most of the work is done in the calls to the BLAS. By contrast, EISPACK is written in Fortran without calls to the BLAS and hence its performance is dependent on the quality of the code generated by the Fortran compiler.

Lehoucq and Carr[35] argue that compilers now have the capability to perform many of the optimizations that the LAPACK project performed by hand. Although no compilers existing today can produce code as efficient as LAPACK from simple three line loops, the compiler technology exists[149, 115, 148].

Today, most compilers are able to produce good code for single loops, reducing the performance advantage of the `BLAS1` routines. Soon compilers will be able to produce good code for `BLAS2` and even `BLAS3` routines. This will require us to rethink certain decisions, especially where the precise functionality that we would like is lacking. There will be an awkward period, probably lasting decades, during which some but not all compilers will be able to perform comparably to the `BLAS`.

2.3.6 Operating Systems

Operating systems are largely irrelevant to serial codes such as `LAPACK` but they can have a significant impact on parallel codes. Consider, for example, the broadcast capability inherent in Ethernet hardware. That capability is not available because the TCP/IP protocol does not allow access to that capability. Furthermore, at least 90% of the message latency cost is attributable to software and the operating system often makes it difficult to reduce the message latency cost. Part of the NOW[3] project involves finding ways to reduce the large message latency cost inherent in Unix operating systems through using user-level to user-level communications, avoiding the operating system entirely.

2.4 Hardware

2.4.1 Processor

The processor, or more specifically the floating point unit, is the fundamental source of processing power or the ultimate limit on performance, depending on your point of view. The combined speed of all of the floating point units is the peak performance, or speed of light, for that computer. For many dense linear algebra codes, the number of floating point operations cannot be reduced substantially and hence the goal is to perform the necessary flops as fast (i.e. as close to the peak performance) as possible.

Floating point arithmetic

The increasing adherence to the `IEEE` standard 754 for binary floating point arithmetic[7] benefits performance in two ways: it reduces the effort needed to make codes work across multiple platforms and it allows one to take advantage of details of the underlying arithmetic in a portable code. The developers of LAPACK had to expend considerable effort to make their codes work on machines with non-IEEE arithmetic, notably older Cray machines. By contrast, the developers of ScaLAPACK chose to concentrate on IEEE standard 754 conforming machines allowing them not only to avoid the hassles of old Cray arithmetic, but also to check the sign bit directly when using bisection[54] to compute the eigenvalues of a tridiagonal matrix.

Consistent floating point arithmetic is also important for execution on heterogeneous machines. Demmel et al.[54] discuss ways to achieve correct results in bisection on a heterogeneous machine. I have proposed having each process compute a subset of eigenvalues, chosen by index, sharing those eigenvalues among all processes and then having each process independently sort the eigenvalues[55].

Ironically the one place where the IEEE standard 754 allows some flexibility has caused problems for heterogeneous machines. The IEEE standard 754 allows several options for handling sub-normalized numbers, i.e. numbers that are too small to be represented as a normalized number. During ScaLAPACK testing it was discovered that a sub-normalized number could be produced on a machine that adheres to the IEEE standard 754 completely and that when this number is then passed to the DEC Alpha 21064 processor, the DEC Alpha 21064 processor does not recognize them as legitimate numbers and aborts. To fix this would have required xdr to be smart enough to recognize this unusual situation[4] or make one of the processors work in a manner different from its default[5].

2.4.2 Memory

The slower speed of main memory (as compared to cache or registers) affects performance in three ways. It reduces the performance of matrix-matrix multiply slightly and greatly complicates the task of coding an efficient matrix-matrix multiply. It bounds from below the algorithmic blocking factor needed to achieve high performance on matrix-matrix multiply. And, it limits the performance of BLAS1 and BLAS2 codes.

The last two factors listed above combine in an unfortunate manner: slow main memory increases the number of BLAS1 and BLAS2 flops and reduces the rate at which they are executed. The number of BLAS1 and BLAS2 flops are typically $O(n^2\,\mathsf{nb})$, where nb is the

[4] This would slow down xdr, possibly significantly.

[5] This too would result in slower execution.

algorithmic blocking factor, which as stated above, must be larger when main memory is slow. The ratio of peak floating point performance to main memory speed is large enough on some machines that the $O(n^2\,\mathsf{nb})$ cost of the BLAS1 and BLAS2 flops can no longer be ignored.

Improving the load balance of the $O(n^2\,\mathsf{nb})$ BLAS1 and BLAS2 flops.

In a blocked dense linear algebra transformation, such as LU decomposition, Cholesky or QR, there are $O(n^2\,\mathsf{nb})$ BLAS1 and BLAS2 flops[30, 53]. PDSYEVX includes two blocked dense linear algebra transformations: Reduction to tridiagonal form, PDSYTRD, is described in Section 4.2 and back transformation, PDORMTR, is described in Section 4.4.

In ScaLAPACK version 1.5, the $O(n^2\,\mathsf{nb})$ BLAS1 and BLAS2 flops are performed by just one row or column of processors. This leads to load imbalance and causes these flops to account for $O(\frac{n^2\,\mathsf{nb}}{\sqrt{p}})$ execution time. If these flops can be performed on all p processors, instead of just one row or column, they will account for only $O(\frac{n^2\,\mathsf{nb}}{p})$ execution time.

There are two ways to spread the cost of the $O(n^2\,\mathsf{nb})$ BLAS1 and BLAS2 flops over all the processors: take them out of the critical path or distribute them over all processors. Transformations such as LU, and back transformation (applying a series of householder vectors) can be pipelined, allowing each processor column (or row) to execute asynchronously. Pipelining in turn allows lookahead, a process by which the active column performs only those computations in the critical path before sending that data on to the next column[32].

Distributing the BLAS1 and BLAS2 flops over all of the processors, as discussed in the last paragraph, requires a different data distribution, a different broadcast and a significant change to the code. The difference can be best illustrated by considering LU. In a 2D blocked LU, LU is first performed on a block of columns, and the resulting LU decomposition is broadcast, or spread across, to all processor columns. One way to broadcast k elements to p processors is to combine a *Reduce_scatter* (which takes k elements and sends k/p to each processor) with an *Allgather* (which takes k/p elements from each processor and spreads them out to all processors giving each processor a copy of all k elements). There are three ways to perform LU on this column block of data: 1) Before the column block is broadcast to all processors (as ScaLAPACK does) in which case only the current column of processors is involved in performing the column LU and the Reduce_scatter and Allgather combine to broadcast the block LU decomposition. 2) After the broadcast, in which case the Reduce_scatter and Allgather combine to broadcast the block column prior to the LU decomposition - all processor columns would have a copy of the block column and each processor column could perform the column block LU redundantly. 3) After the Reduce_scatter but before the Allgather. In this case, the Reduce_scatter operates on the column block prior to the LU decomposition but the Allgather operates on the block column after the LU decomposition. All processors can be involved in the LU decomposition.

In `HJS`, Hendrickson, Jessup and Smith's symmetric eigensolver[91, 154] discussed in Section 7.1.2, the `BLAS1` and `BLAS2` flops are analogously distributed over all of the processors.

Lookahead does not improve performance unless the execution of the code is pipelined, i.e. proceeds in a wave pattern over the processes. Two-sided reductions, like tridiagonal reduction, do not allow pipelining. And, pipelining may be limited on reductions of symmetric or Hermitian matrices (such as Cholesky)[6].

Memory size

The amount of main memory limits the size of the problem that can be executed efficiently, while the amount of virtual memory limits the size of the problem that can be run at all. `ScaLAPACK`'s symmetric eigensolvers, `PDSYEVX` and `PDSYEV` require roughly $4n^2$ and $2n^2$ double precision words of virtual memory space respectively. However, both can be run efficiently provided that physical memory can contain[7] the $n^2/2$ elements of the triangular matrix A. Ed D'Azevedo[52] has written an out-of-core symmetric eigensolver for `ScaLAPACK` and studied the performance of `PDSYEV` and `PDSYEVX` on large problem sizes.

2.4.3 Parallel computer configuration

I will discuss primarily distributed memory computers with one processor per node, discussing shared memory computers (SMPs), clusters of workstations and clusters of shared memory computers only briefly.

Four machine characteristics are important for distributed memory computers: peak floating point performance, software overhead, communication latency and commu-

[6] I believe that pipelining can be used in Cholesky if a square processor grid is used. Work in progress.

[7] Depending on the page size, keeping an n by n triangular matrix in memory may require as few as $n^2/2$ memory locations (if the page size is 1) or as many as n^2 (if the page size is $\geq n$).

nication (bisection) bandwidth. Software overhead and communication latency are the dominant costs for small problems[8]. Peak floating point performance is the dominant costs for large problems [9].

Interconnection network

Bisection bandwidth and communication latency are the two important measures of an interconnection network. Networks which allow only one pair of nodes to communicate at a time do not offer adequate bisection bandwidth and hence parallel dense linear algebra (with the possible exception of huge matrix-matrix multiplies) will not perform well on such a network.

As long as the bisection bandwidth is adequate, the topology of the interconnection network has not proven to be an important factor in the performance of parallel dense linear algebra.

Shared Memory Multiprocessing

Users of dense linear algebra codes have two choices on shared memory multiprocessors. They can use a serial code, such as `LAPACK` that has been coded in terms of the `BLAS` and, provided that the manufacturer has provided an optimized `BLAS`, they will achieve good performance. Or, provided that the manufacturer provides `MPI`[65], `PVM`[19] or the `BLACS` they can use `ScaLAPACK`.

LeBlanc and Markatos[118] argue that shared memory codes typically get better load balance while message passing codes typically incur lower communications cost. However, the real difference could well come down to a matter of how efficient the underlying libraries are.

Clusters of workstations

Some clusters of workstations, notably the NOW project[3] at Berkeley, offer comparable communication performance to distributed memory computers. However, the vast majority of networks of workstations in present use are still connected by Ethernet or FDDI

[8] On current architectures, $n < 100\sqrt{p}$ is small for our purposes

[9] On current architectures, $n > 1000\sqrt{p}$ is large for our purposes

rings and hence do not have the low latency and high bisection bandwidth required to perform dense linear algebra reductions in parallel efficiently.

Cluster of SMPs (CLUMPS)

Dense linear algebra codes have two choices on clusters of SMPs: they can assign one process to each processor or they can assign one process to each multi-processor node. The tradeoff will be similar to the shared-memory versus message-passing question on shared memory computers.

If each processor is assigned a separate process the details of how the processes will be assigned to what is essentially a two level grid of processors will be important. For a modest cluster of SMPs (say 4 nodes each with 4 processors) it might make sense to assign one dimension within the node and the other across the nodes. However, this will not scale well - adding nodes will require increasing the bandwidth per node else all dense linear algebra transformations will become bandwidth limited as the number of nodes increases. A layout which that is 2 dimensional within the nodes and 2 dimensional among the nodes allows both the number of processors per node and the number of nodes to increase provided only that bisection bandwidth grow with the number of processors and that internal bisection bandwidth (i.e. main memory bandwidth) grows with the number of processors per node.

On the first CLUMPS, how well each of the libraries is implemented is likely to outweigh theoretical considerations. Shared memory `BLAS` are not trivial, nor will communication systems that properly handle two levels of processor hierarchy be, i.e. communication within a node and communication between nodes.

On most distributed memory systems, the logical to physical processor grid mapping is of secondary importance. I suspect that this will not be the case for clusters of SMPs. It will be important to have the processes assigned to the processors on a particular node nearby in the logical process grid as well.

2.5 Applications

Large symmetric eigenproblems are used in a variety of applications. Some of these applications include: real-time signal processing[156] [34], modeling of acoustic and electro-magnetic waveguides[114], quantum chemistry[74] [22, 175], numerical simulations of disordered electronic systems[95], vibration mode superposition analysis[18], statistical mechanics[132], molecular dynamics[152], quantum Hall systems[112, 106], material science[166], and biophysics[143, 144].

The needs of these applications differ considerably. Many require considerable execution time to build the matrix and hence the eigensolution remains a modest part of the total execution. However, building the matrix often parallelizes easily and grows much more slowly than the $O(n^3)$ cost of eigensolution. Hence, for these applications, the eigensolver becomes the bottleneck as larger problems are solved in parallel. Few applications require the entire spectrum, but most of these listed above require at least 10% of the spectrum and hence are best solved by dense techniques. Some have large clusters of eigenvalues[74], while others do not.

2.5.1 Input matrix

Three features of the input matrix affect the execution time of symmetric eigensolvers: sparsity, eigenvalue clustering and spectral diagonal dominance.

Sparsity

Some algorithms and codes are specifically designed for sparse input matrices. Lanczos[49] has traditionally been used to find a few eigenvalues and eigenvectors at the ends of the spectrum. Recently, ARPACK[119], and PARPACK[130] have been developed based on Lanczos with full re-orthogonalization. They can therefore compute as much of the spectrum as the user chooses.

The Invariant Subspace Decomposition Approach and reduction to tridiagonal form based algorithms can both be run from either a dense or banded matrix. In this dissertation, I discuss only dense matrices.

Spectrum

Some algorithms are more dependent on the spectrum than others. Most are dependent in some manner, but that dependence differs from one algorithm to another.

It is difficult to maintain orthogonality of the eigenvectors when computing the eigendecomposition of matrices with tight clusters of eigenvalues. Such matrices require special techniques in divide and conquer and in inverse iteration (See section 2.7.4). On the other hand, divide and conquer experiences the most deflation, and hence the greatest efficiency, on matrices with clustered eigenvalues.

The Invariant Subspace Decomposition Approach maintains orthogonality on matrices with clustered eigenvalues. However, it may have difficulty picking a good split point if the clustering causes the eigenvalues to be unevenly distributed.

Spectral Diagonal dominance

Spectral diagonal dominance[10] speeds convergence of the Jacobi algorithm. Indeed, if the input matrix is sufficiently diagonally dominant, Jacobi may converge in as little as two steps (versus 10 to 20 for non diagonally dominant matrices). But, spectral diagonal dominance has little effect on any of the other algorithms.

2.5.2 User request

The portion of the spectrum that the user needs, i.e. the number of eigenvalues and/or eigenvectors, affects execution time of some, but not all eigensolvers.

Two step band reduction (to tridiagonal form) is most attractive when only eigenvalues are requested because the back transformation task is expensive in two step band reduction.

The cost of bisection and inverse iteration depends upon the number of eigenvalues and eigenvectors requested. These costs are $O(n^2)$ and generally not significant for large problem sizes. However, back transformation requires $2n^2m$ flops where m is the number of eigenvectors required.

Iterative methods, such as Lanczos[49] and implicitly restarted Lanczos[119] are clearly superior if only a few eigenvectors are required.

[10] Spectrally diagonally dominant means that the eigenvector matrix, or a permutation thereof, is diagonally dominant. Most, but not all, diagonally dominant matrices are spectrally diagonally dominant. For example if you take a dense matrix with elements randomly chosen from $[-1, 1]$ and scale the diagonal elements by $1e3$ the resulting diagonally dominant matrix will be spectrally diagonally dominant. However, if you take that same matrix and add $1e3$ to each diagonal element, the eigenvector matrix is unchanged even though the matrix is clearly diagonally dominant.

2.5.3 Accuracy and Orthogonality requirements.

Demmel and Veselić,[58] prove that on scaled diagonally dominant matrices[11], Jacobi can compute small eigenvalues with high relative accuracy while tridiagonal based methods can fail to do so.

At present, the ScaLAPACK offers two symmetric eigensolvers: PDSYEVX and PDSYEV. PDSYEVX, which is based on bisection and inverse iteration (DSTEBZ and DSTEIN from LAPACK) is faster and scales better but does not guarantee orthogonality among eigenvectors associated with clustered eigenvalues. PDSYEV, which is based on QR iteration (DSTEQR from LAPACK) is slower and does not scale as well but does guarantee orthogonal eigenvectors.

2.5.4 Input and Output Data layout

At present, the execution time of the ScaLAPACK symmetric eigensolver is strongly dependent on the data layout chosen by the user for input and output matrices. 1D data layouts are not scalable and lead to both high communication costs and poor load balancing. Suboptimal block sizes can likewise affect performance significantly. In particular, a block size of 1, i.e. cyclic data layout, causes ScaLAPACK to send a large number of small messages resulting in unacceptable message latency costs and a huge number of calls to the BLAS. If the block size is too large, load balance suffers.

There are a couple ways to reduce this dependence on the data layout chosen by the user. If algorithmic blocking is separated from data layout blocking[140] [91] [159] small data layouts can be handled much more efficiently. However, small block-sizes (especially cyclic layouts) still require more messages than larger block-sizes. And, large block sizes still lead to load imbalance.

In Chapter 8 I will show that redistributing the data to an internal format that is near optimal for the particular machine and algorithm involved allows for improved performance and performance that is independent of the input and output data layout.

2.6 Machine Load

The load of the machine, in addition to the direct effect of offering your program only a portion of the total cycles, can have several indirect effects. If each processor is

[11]A matrix, A, is scaled diagonally dominant if and only if DAD with $D = |diag(A)|^{1/2}$ is diagonally dominant.

individually scheduled, performance can be arbitrarily poor because significant progress is only possible when all processes are concurrently scheduled. A loaded machine may also cause your data to be swapped out to disk, which can greatly reduce peak performance. Finally, it is the most heavily loaded machine which controls execution time. If your code is running on 9 unloaded processors and one processor with a load factor of 5, you will get no more than a factor of 10/5 speedup. A ScaLAPACK user has reported performance degradation and speedup less than 1, (i.e. more processors take longer to comlete the same sized eigendecomposition) on the IBM IBM SP2. I have also witnessed this behaviour on the IBM IBM SP2at the University of Tennesse at Knoxville and I have reason to suspect that the IBM IBM SP2 isnot gang scheduled and that this fact accounts for a large part of the poor performance of PDSYEVX that the user and I have witnessed on the IBM SP2.

Space sharing, allocating subsets of the processors, solves all of these problems, but has its own problems. On some machines, jobs running on different partitions share the same communications paths and hence if one job saturates the network, all jobs may suffer.

2.7 Historical notes

2.7.1 Reduction to tridiagonal form and back transformation

Householder reduction to tridiagonal form is a two-sided reduction, which requires multiplication by Householder reflectors from both the left and right side. Martin et al. implemented reduction to tridiagonal form in Algol[129]. TRED1 and TRED2 perform reduction to tridiagonal form in EISPACK[153]. Dongarra, Hammarling and Sorensen[64] showed that Householder reduction to tridiagonal form can be performed using half matrix-vector and half matrix-matrix multiply flops. This has been implemented as DSYTRD in LAPACK[5, 67] for scalar and shared memory multiprocessors and PDSYTRD for distributed memory computers in ScaLAPACK[42]. Chang et al. implemented one of the first parallel codes for reduction to tridiagonal form, first using a 1D cyclic data layout[37] and then a 2D cyclic data layout[38].

Smith, Hendrickson and Jessup[91] show that data blocking is not required for efficient algorithmic blocking and that PDSYTRD pays a substantial execution time penalty for its generality (accepting any processor layout) and portability (being built on top of the PBLAS, BLACS and BLAS). By restricting their attention to square processor layouts on the PARAGON, they were able to dramatically reduce the overhead incurred in reduction to tridiagonal form in HJS. HJS does not have the redundant communication found in PDSYEVX, it makes many fewer BLAS calls, avoids the overhead of the PBLAS calls, and spreads the work more evenly among all the processors (improving load balance). Furthermore, HJS, by using communication primitives better suited to the task, reduces both the number of messages sent and the total volume of communication substantially. Some, but not all, of these advantages necessitate that the processor layout be square. HJS is discussed in Section 7.1.2.

Other ways to reduce the execution time of reduction to tridiagonal form do not require that the processor layout be square. Bischof and Sun[25] and Lang[116] showed that in a two step band reduction to tridiagonal form, all of the flops, asymptotically, can be performed in matrix multiply routines. Karp, Sahay, Santos and Schauser[107] showed that subset broadcasts and reductions can be performed optimally. Van de Geijn and others[16] are working to implement improved subset broadcast and reduction primitives.

Hegland et al.[90] argue that the fastest way to reduce a symmetric matrix A to tridiagonal form on the VPP500 (a multiprocessor vector supercomputer by Fujitsu) is to compute $L_1 D L_1^T = A$ and then compute a series of L_i using orthonormal transformations such that $L_{n+p-1} D L_{n+p-1}^T$ is tridiagonal. Their technique is, in essence, a two step band reduction in which the two steps are performed within the same loop. Let $L_i[:, \mathsf{own}(\rho)]$ represent the columns of L_i, owned by processor ρ. ${}_\rho Q_i$ means the portion of Q_i which processor ρ owns.

The code is:

$L_1 D L_1^T = A$

For $i = 1$ to $n-1$ do:

  Each processor independently performs:

    ${}_\rho Q_i = \mathsf{House}(L_i[:, \mathsf{own}(\rho)] D_i[\mathsf{own}(\rho), \mathsf{own}(\rho)] L_i[:, \mathsf{own}(\rho)]^T)$

    $L_{i+1}[:, \mathsf{own}(\rho)] =_\rho Q_i L_i[:, \mathsf{own}(\rho)]$

  The processors together perform:

    $\mathsf{Allgather}(L_{i+1}[:, i+1 : i+p])$

  Each processor performs redundantly:

    $Q_i' = \mathsf{House}(L_{i+1}[:, i+1 : i+p] D[i+1 : i+p, i+1 : i+p] L_{i+1}[:, i+1 : i+p]^T)$

$$L_{i+1}[:, i+1 : i+p] = Q_i' L_{i+1}[:, i+1 : i+p]$$

In Allgather($L_{i+1}[:, i+1 : i+p]$) each processor contributes the column of $L_{i+1}[:, i+1 : i+p]$ which it owns and all processors end up with identical copies of $L_{i+1}[:, i+1 : i+p]$.

The loop invariants are as follows:

$$\text{Let: } T_i = (L_i)D(L_i)^T$$

$$\forall_{j<i;k<i+p} T_i(j,k) = 0 \qquad \text{(Line 1)}$$

$$T_i(1 : i-p, 1 : i-p) \text{is tridiagonal} \qquad \text{(Line 2)}$$

For $p = 1$, the serial case, both of these conditions are identical and meeting them requires computing the first column of $(L_i)D(L_i)^T$, computing the Householder vector and applying it to L_i to yield L_{i+1}.

For $p > 1$, the parallel case, the first loop invariant is maintained by each processor independently computing the first column of $(L_i)D(L_i)^T$, using only the local columns[12] of L_i. A Householder vector is computed from this and applied to the local columns of L_i. The second loop invariant is maintained redundantly on all processors. All processors obtain copies of columns i to $i+p-1$ of L_i and compute: $A(1 : p, 1) = L_i(i : i+p-1, i : i+p-1)D(i : i+p-1, i : i+p-1)L(i : i+p-1, i)^T$. A Householder vector is computed from $A(1 : p, 1)$ and applied to $L_i(i : i+p-1, :)$, redundantly on all processors, maintaining the second loop invariant.

This one-sided transformation requires fewer messages than Hessenberg reduction to tridiagonal form and, for small p, less message volume, but requires twice as many flops.

2.7.2 Tridiagonal eigendecomposition

Sequential symmetric QL and QR algorithms

The implicit QL or QR algorithms have been the most commonly used methods for solving the symmetric eigenproblem for the last couple decades. Francis[79] wrote the first implementation of the QL algorithm based on Rutishauser's LR transformation. The QL algorithm is the basis of the `EISPACK` routine `IMTQL1`, while the `LAPACK` routine `DSTEQR` uses either implicit QR or implicit QL depending on the top and bottom diagonal elements[86].

[12] Their implementation uses a column cyclic data distribution.

Henry[93] shows that if between each sweep of QR (or QL) in which the eigenvectors are updated an additional sweep is performed in which the eigenvectors are not updated, better shifts can be used, reducing the total number of flops from roughly $6n^3$ to $4n^3$.

Reinsch[145] wrote `EISPACK`'s `TQLRAT` which computes eigenvalues without square roots. `LAPACK`'s `DSTERF` improves on `TQLRAT` using a root free variant developed by Pal, Walker and Kahan[134]. Like `DSTEQR`, `DSTERF` uses either implicit QR or implicit QL depending on the top and bottom diagonal elements

Parallel symmetric QL and QR algorithms

QR requires $O(n^2)$ effort to compute the eigenvalues and $O(n^3)$ to compute the eigenvectors. No one has found a good, stable way to parallelize the $O(n^2)$ cost of computing the eigenvalues and reflectors. Sameh and Kuck[113] use parallel prefix to parallelize QR for eigenvalue extraction. They obtain $O(\frac{1}{\log(p)})$ speedup, but they do not show how their method can be used to generate reflectors and hence eigenvectors.

However, parallelizing the $O(n^3)$ effort of computing the eigenvectors is straightforward as shown by Chinchalkar and Coleman[39]; and Arbenz et al.[8] and implemented for `ScaLAPACK` by Fellers[76].

Symmetric QR parallelizes nicely in a MIMD programming style, but efforts to parallelize it on a shared memory machine in which the parallelism is strictly within the calls to the `BLAS` have produced only modest speedups. Bai and Demmel[13] first suggested using multiple shifts in non-symmetric QR. Arbenz and Oettli[10] showed that blocking and multiple shifts could be used to obtain modest improvements in the speed (roughly a factor of 2 on 8 processors) of QR for eigenvalues and eigenvectors on the ALLIANT FX/80. Kaufman[109] showed that multi-shift QR could be used to speed eigenvalue extraction by a factor of 3 on a 2-processor Cray YMP despite tripling the number of flops performed.

Sturm sequence methods

Givens[83] used bisection to compute the eigenvalues of a tridiagonal matrix based on Wilkinson's original idea. Kahan[105] showed that bisection can compute small eigenvalues with tiny componentwise relative backward error, and sometimes high relative accuracy. High relative accuracy is required for inverse iteration on a few matrices. Barlow and Evans were the first to use bisection in a parallel code[15].

Computing eigenvalues of a tridiagonal matrix can be split into three phases: isolation, separation and extraction. The isolation phase identifies, for each eigenvalue, an interval which contains that eigenvalue and no other. The separation phase improves the eigenvalue estimate. And the extraction phase computes the eigenvalue to within some tolerance. Bisection can be used for all three phases.

Neither existing codes, nor the literature explicitly distinguish between these three phases, but they have very different computational aspects. Isolation, at least to the point of identifying p intervals so that each processor is responsible for one interval is difficult to parallelize, whereas the other phases are fairly straightforward. The separation phase is typically the challenge for most root finders, and the area where they distinguish themselves from other codes. Divide and conquer techniques which use the eigenvalues from perturbed matrices as estimates of the eigenvalues of the original matrix, isolate and may separate the roots.

Techniques for eigenvalue isolation include: multi-section[126] [14], assigning different parts of the spectrum to different processors[95, 20], divide and conquer and using multiple processors to compute the inertia of a tridiagonal matrix[123]. In multi-section, each processor computes the inertia at a single point, splitting an interval into $p+1$ intervals. Although multi-section requires communication, Crivelli and Jessup[48] show that the communication cost is often a modest part of the total cost. Divide and conquer splits the matrix by perturbing or ignoring a couple of elements, typically near the center of the matrix to separate the matrix into two tridiagonal matrices whose eigenvalues can be computed separately. If a rank 1 perturbation is chosen, the merged set of eigenvalues provides a set of intervals in which exactly one eigenvalue lies.

There are a number of ways to use multiple processors to compute the inertia of a tridiagonal matrix. Lu and Qiao[127] discuss using parallel prefix to compute the Sturm sequence as the sub-products of a series of 2 by 2 matrices and Mathias[131] did an error analysis and showed that it was unstable. Ren[146] tried unsuccessfully to repair parallel prefix. Conroy and Podrazik[46] perform LU on a block arrowhead matrix. Each block is tridiagonal and the arrow has width equal to the number of blocks. Swarztrauber[162] and Krishnakumar and Morf[111] discuss ways of computing the determinant of 4 matrices of size roughly n by n from the determinants of 8 matrices of size roughly $\frac{1}{2}n$ by $\frac{1}{2}n$. Each of these methods performs 2 to 4 times more floating point operations than a serial Sturm sequence count would and requires $O(\log(p))$ messages. Except for Conroy and Podrazik's method, they all use multiplies instead of divides. Multiplies are faster than divides, but require special checks to avoid overflow.

The computation of the inertia is slowed by the existence of a divide and a comparison in the inner loop. There are also a couple tricks that can potentially be used to speed computation of the inertia to reduce the number of divides and comparisons or to make them faster. `ScaLAPACK`'s `PDSYEVX` uses signed zeroes and the C language ability to extract the sign bit of a floating point number to avoid a comparison in the inner loop[54]. I have proposed perturbing tiny entries in the tridiagonal matrix to guarantee that negative zero will never occur, thus allowing a standard C or Fortran comparison against zero. Using a standard comparison against zero would allow compilers to produce more efficient code. I have also proposed reducing the number of divides in the inner loop by taking advantage of the fixed exponent and mantissa sizes in `IEEE` double precision numbers. I have not implemented either of these ideas. Some machines have two types of divide: a fast hardware divide that may be incorrect in the last couple bits and a slower but correct software divide. Demmel, Dhillon and Ren[54] give a proof of correctness for `PDSTEBZ`, `ScaLAPACK`'s bisection code for computing the eigenvalues of a tridiagonal matrix, in the face of heterogeneity and non-monotonic arithmetic (such as sloppy divides). This shows that bisection can be robust even in the face of incorrect divides.

Many techniques that have been used to accelerate eigenvalue extraction including: the secant method[33], Laguerre's iteration[138], Rayleigh quotient iteration[163], secular equation root finding[50] and homotopy continuation[120, 45]. Bassermann and Weidner use a Newton-like root finder called the Pegasus method[17]. These acceleration techniques converge super-linearly as long as the eigenvalues are separated.

Li and Ren[121] accelerate eigenvalue separation in their Laguerre based root finder by detecting linear convergence and estimating the effect of the next several steps. Brent[33] discusses ways of separating eigenvalues when the secant method is used. Li and Zeng use an estimate of the multiplicity in their root finder based on Laguerre iteration[122]. Szyld[163] uses inverse iteration with a shift set to middle of the interval known to contain only one eigenvalue to separate eigenvalues before switching to Rayleigh quotient iteration. Cuppen's method takes advantage of multiple eigenvalues through deflation.

Eigenvalue extraction can be performed in parallel with no communication, or a small constant amount of communication. However, eigenvalue extraction can exhibit poor load balance, especially if acceleration techniques are used. Ma and Szyld[128] use a task queue to improve load balance. Li and Ren[121] minimize load imbalance by concentrating on worst case performance.

ScaLAPACK chose bisection and inverse iteration for its first tridiagonal eigensolver, PDSYEVX, because they are fast, well known, robust, simple and parallelize easily. ScaLAPACK has since added a QR based tridiagonal eigensolver for those applications needing guarantees on orthogonality within eigenvectors corresponding to large clusters of eigenvalues. See section 4.3 for details.

Divide and Conquer

Cuppen[50] showed that by making a small perturbation to a tridiagonal matrix it could be split into two separate tridiagonal matrices each of which could be solved independently, and that the eigendecomposition of the original tridiagonal matrix could then be constructed from the eigendecomposition of the two independent tridiagonal matrices and the perturbation.

There are many ways to perturb a tridiagonal matrix such that the result is two separate tridiagonal matrices. The following four have been implemented. Cuppen's algorithm[50] subtracts $\alpha u u^T$ from the tridiagonal matrix, where $u = e_{\frac{1}{2}n} + e_{\frac{1}{2}n+1}$ and $\alpha = T_{\frac{1}{2}n,\frac{1}{2}n+1}$. Gu and Eisenstat[89] set all elements in row and column i to zero. Gates and Arbenz[82] call this a rank-one extension and refer to this as permuting row and column $\frac{1}{2}n$ to the last row and column (as opposed to setting all elements in row and column i to zero). Gates[80] uses a rank two perturbation: $T_{\frac{1}{2}n,\frac{1}{2}n+1}(e_{\frac{1}{2}n}e^T_{\frac{1}{2}n+1} + e_{\frac{1}{2}n+1}e^T_{\frac{1}{2}n})$ is subtracted from the original tridiagonal.

Cuppen's original divide and conquer method can result in a loss of orthogonality among the eigenvectors. Three methods of maintaining orthogonality have been implemented. Sorensen and Tang[155] calculate the roots to double precision. Gu and Eisenstat[89] compute the eigenvectors to a slightly perturbed problem. Gates[81] showed that inverse iteration and Gram-Schmidt re-orthogonalization could be used in divide and conquer codes to compute orthogonal eigenvectors.

Several divide and conquer codes are available today. The first publically available divide and conquer code, TREEQL was written by Dongarra and Sorensen[66]. The fastest reliable serial code currently available for computing the full eigendecomposition of a tridiagonal matrix is LAPACK's DSTEDC[147]. It is based on Cuppen's divide and conquer[50] and uses Gu and Eisenstat's[88] method to maintain orthogonality.

There has long been interest in parallelizing divide and conquer codes because of the obvious parallelism involved in the early stages. There are three reasons why this technique has proven difficult to parallelize. The first is that the majority of the flops are performed at the root of the divide and conquer tree and hence the parallelism at the leaves is less valuable[36]. The second is that deflation, the property that makes `DSTEDC` the fastest serial code, leads to dynamic load imbalance in parallel codes. The third is the complexity of the serial code itself.

Dongarra and Sorensen's parallel code[66], `SESUPD`, was written for a shared memory machine. The first parallel divide and conquer codes written for distributed memory computers used a 1D data layout (thus limiting their scalability)[99, 81]. Potter[141] has written a parallel divide and conquer for small matrices (it requires a full copy of the matrix on each node). Françoise Tisseur has written a parallel divide and conquer code for inclusion in `ScaLAPACK`.

Inverse Iteration

Inverse iteration with eigenvalue shifts is typically used to compute the eigenvectors once the eigenvalues are known[170]. Jessup and Ipsen[102] explain the use of Gram-Schmidt re-orthogonalization to ensure that the eigenvectors are orthogonal. Fann and Littlefield[75] found that inverse iteration and Gram-Schmidt can be performed in parallel, greatly improving its efficiency. Parlett and Dhillon[139, 59] are working on a method, based on work by Fernando, Parlett and Dhillon[77], that may avoid, or greatly reduce the need for re-orthogonalization.

The Jacobi method

The Jacobi method for the symmetric eigenproblem consists of applying a series of rotators each of which forces a single off-diagonal element to zero. Each such rotation reduces the square of the Frobenius norm of the off-diagonal elements by the square of the element which was eliminated. Hence, as long as the off-diagonal elements to be eliminated are reasonably chosen, the norm of the off-diagonal converges to zero[167].

There are several variations in the Jacobi method. Classical Jacobi[100], selects the largest off-diagonal element as the element to eliminate at each step, and hence requires the fewest steps. However, $O(n^2)$ comparisons are required at each step to select the largest element, requiring $O(n^4)$ comparisons per sweep, rendering it unattractive. Cyclic Jacobi annihilates every element once per sweep in some specified order. Threshold Jacobi differs from cyclic Jacobi in that only those elements larger than a given threshold are annihilated. Block Jacobi annihilates an entire block of elements at each step.

Cyclic, threshold and block variants of Jacobi each have their advantages. Cyclic Jacobi is the simplest to implement. Block Jacobi requires fewer flops (and if done in parallel, fewer messages) per element annihilated. Threshold Jacobi requires fewer steps and converges more surely than cyclic Jacobi, however a parallel threshold Jacobi requires more communication. Scott et al. showed that a block threshold Jacobi method[151] is the best Jacobi method for distributed memory machines, however, it would also be the most complex to implement. Littlefield and Maschhoff[125] found that for large numbers of processors, a parallel block Jacobi beat tridiagonal based methods available at that time.

One-sided Jacobi methods apply rotations to only one side of the matrix and force the columns of the matrix to be orthogonal, hence represent scaled eigenvectors. One-sided Jacobi methods require fewer flops and may parallelize better[10, 21].

Existing parallel implementations of the Jacobi algorithm are based on a 1D data layout. Arbenz and Oettli[10] implemented a blocked one-sided Jacobi. Pourzandi and Tourancheau[142] show that overlapping communication and computation is effective in a Jacobi implementation on the i860 based NCUBE. Although a 1D data layout is not scalable, the huge computation to communication ratio in the Jacobi algorithm hides this on all machines available today.

There are two publically available parallel Jacobi codes. Fernando wrote a parallel Jacobi code for `NAG`[87]. O'Neal and Reddy[133] wrote a parallel Jacobi, `PJAC`, for the Pittsburgh Supercomputing Center.

Demmel and Veselić,[58] prove that on scaled diagonally dominant matrices, Jacobi can compute small eigenvalues with high relative accuracy while tridiagonal based methods cannot. Demmel et al.[56] give a comprehensive discussion of the situations in which Jacobi is more accurate than other available algorithms.

The Jacobi method is discussed in Section 7.3.

2.7.3 Matrix-matrix multiply based methods

There are several methods for solving the symmetric eigenproblem which can be made to use only matrix-matrix multiply.

Matrix-matrix based methods are attractive because they can be performed efficiently on all computers, and they scale well. However, they require many more flops (typically 6 - 60 times more) than reduction to tridiagonal form, tridiagonal eigensolution and back transformation. Hence, these methods only make sense if tridiagonal based methods cannot be performed efficiently or do not yield answers that are sufficiently accurate.

Invariant Subspace Decomposition Algorithm

The Invariant Subspace Decomposition Algorithm[97], ISDA, for solving the symmetric eigenproblem involves recursively decoupling the matrix A into two smaller matrices. Each decoupling is achieved by applying an orthogonal similarity transformation, $Q^T AQ$, such that the first columns of Q span an invariant subspace of A. Such a Q is found by computing a polynomial function of A, $A' = p(A)$ which maps all the eigenvalues of A nearly to 0 or 1, and then taking the QR decomposition of p(A). One such polynomial can be computed by first shifting and scaling A so that all its eigenvalues are known to be between 0 and 1 (by Gershgorin's theorem) and then repeatedly computing the beta function, $A_{i+1} = 3A_i^2 - 2A_I^3$, until all of the eigenvalues of A_i are effectively either 0 or 1. (All of the eigenvalues of A_0 that are less than 0.5 are mapped to 0, all the eigenvalues of A_0 that are greater than 0.5 are mapped to 1.)

The ISDA parallelizes well because each of the tasks involved perform well in parallel[97]. Unfortunately, the ISDA requires far more floating point operations (roughly 100 n^3) than eigensolvers that are based on reducing the matrix first to tridiagonal form (which require $8n^3 + O(n^2)$ or fewer flops).

Applying the ISDA for banded matrices greatly reduces the flop count[26]. Furthermore, the banded matrix multiplications can still be performed efficiently, and the bandwidth does not triple with each application of $A_{i+1} = 3A_i^2 - 2A_I^3$ as one would expect with random banded matrices. Nonetheless, the bandwidth does grow enough to necessitate several band reductions, each of which requires a corresponding back transformation step.

A publically available code based on the ISDA is available from the PRISM group[28].

The ISDA applied directly to the full matrix requires roughly $100n^3$ flops, or 30 times as many as tridiagonal reduction based methods, and hence will never be as fast. Banded ISDA is almost a tridiagonal based method, but is not likely to be the fastest method. The quickest way to compute eigenvalues from a banded matrix is to reduce the matrix first to tridiagonal form. And, if eigenvectors are required, banded ISDA will require at least twice and probably three times as many flops in back transformation.

FFT based invariant subspace decomposition

Yau and Lu[174] implemented an FFT based invariant subspace decomposition method. This method requires $O(\log(n))$ matrix multiplications. Tisseur and Domas[60] have written a parallel implementation of the Yau and Lu method.

FFT based invariant subspace decomposition, like ISDA applied to dense matrices requires roughly $100n^3$ flops. Hence, it, like ISDA will never be as fast as tridiagonal reduction based methods.

Strassen's matrix multiply

Strassen's matrix-matrix multiply[157] can decrease the execution time for very large matrix-matrix multiplies by up to 20% but will not make ISDA competitive. Several implementations of Strassen's matrix multiply have been able to demonstrate performance superior to conventional matrix-matrix multiply[96][43]. However, Strassen's method is only useful when performing matrix-matrix multiplies in which all three matrices are very large and Strassen's flop count advantage grows very slowly as the matrix size grows. In order to double Strassen's flop count advantage, the matrices begin multiplied must be sixteen times as large and hence memory usage must increase a thousand fold.

2.7.4 Orthogonality

Some methods, notably inverse iteration, require extra care to ensure that the eigenvectors are orthogonal. In exact arithmetic, if two eigenvalues differ, their corresponding eigenvectors will be orthogonal. However, if the input matrix has, say, a double eigenvalue, the eigenvectors corresponding to this double eigenvalue span a two-dimensional subspace and hence there is no guarantee that two eigenvectors chosen at random from this space will be orthogonal. In floating point arithmetic, inverse iteration without re-orthogonalization may not produce orthogonal eigenvectors when two or more eigenvalues are nearly identical. In DSTEIN, LAPACK's inverse iteration code, when computing the eigenvectors for a cluster of eigenvalues, modified Gram-Schmidt re-orthogonalization is employed after each iteration to re-orthogonalize the iterate against all of the other eigenvalues in the cluster[102]. Modified Gram-Schmidt re-orthogonalization parallelizes poorly because it is a series of dot products and DAXPY's each of which depends upon the result of the immediately preceding operation. PeIGs[74] and PDSYEVX[68] have chosen different responses to the fact that the re-orthogonalization in DSYEVX parallelizes poorly.

PeIGs alternates inverse iteration and re-orthogonalization in a different manner than DSYEVX. Instead of computing one eigenvector at a time, all of the eigenvectors within a cluster are computed simultaneously. For each cluster, PeIGs first performs a round of inverse iteration without re-orthogonalization using random starting vectors. Then, PeIGs performs modified Gram-Schmidt re-orthogonalization twice to orthogonalize the eigenvectors. PeIGs performs a second round of inverse iteration without re-orthogonalization, using the output from the previous step as the starting vectors, and again repeating until sufficient accuracy is obtained for each eigenvector. Finally, PeIGs performs modified Gram-Schmidt re-orthogonalization one last time. They have shown that this method works on application matrices with large clusters of eigenvalues.

PDSYEVX attempts to assign the computation of all eigenvectors associated with each cluster of eigenvalues to a single processor. When enough space is available to accomplish this, PDSYEVX produces exactly the same results as DSYEVX. When the user does not provide enough local workspace PDSYEVX relaxes the definition of cluster repeatedly until it can assign all the computation of all eigenvectors associated with each cluster of eigenvalues to a single processor.

When the input matrix contains one or more very large clusters of eigenvalues, PDSYEVX performs poorly: If enough workspace is available, PDSYEVX gives the same results as DSYEVX, but runs very slowly. If insufficient workspace is available, PDSYEVX does not guarantee orthogonality. Dhillon explains the fundamental problems in inverse iteration[59].

Recently Parlett and Dhillon have identified new techniques for computing the eigenvectors of a symmetric tridiagonal matrix[136, 139]. These new results raise the hope that we will soon have an $O(n^2)$ method for computing the eigenvectors of a symmetric tridiagonal matrix which parallelizes well and avoids the problems with computing the eigenvectors associated with clustered eigenvalues. ScaLAPACK looks forward to applying these new techniques in a future release.

Chapter 3

Basic Linear Algebra Subroutines

3.1 BLAS design and implementation

The BLAS[117, 63, 62], Basic Linear Algebra Subroutines, were designed to allow portable codes most of whose operations are matrix-matrix multiplications, matrix-vector multiplications, and related linear algebra operations to achieve high performance provided that the BLAS achieve high performance. In LAPACK[4], the BLAS were used to re-express the linear algebra algorithms in the previous libraries Linpack[61] and EISPACK[153], thereby achieving performance portability.

The BLAS routines are split into three sets. BLAS Level 1 routines involve only vectors, require $O(n)$ flops (on input vectors of length n) and two or three memory operations for every two flops performed. BLAS Level 2 routines involve one n by n matrix, $O(n^2)$ flops and one or two memory operations for every two flops performed (rectangular matrices are also supported). BLAS Level 3 routines involve only matrices, $O(n^3)$ flops and $O(n^2)$ memory operations. BLAS Level 1, because they involve only $O(n)$ operations per invocation, have the least flexibility in how the operations are ordered, and require the most memory operations per flop. Hence, BLAS Level 1 routines have the lowest peak floating point operation rate. They also have the lowest software overhead - an important consideration because they perform few operations. BLAS Level 3 routines have the most flexibility in how the operations are ordered and require the fewest memory operations per flop and hence achieve the highest performance on large tasks. BLAS Level 1 and 2 routines are typically limited by the speed of memory. BLAS Level 3 routines typically execute very near the peak speed of the floating point unit.

Typical hardware architectures make it possible, but not easy, to achieve high floating point execution rates for matrix-matrix multiply. Floating point units can initiate floating point operations every 2 to 5 nanoseconds though floating point operations take 10 to 30 nanoseconds to complete and main memory requires 20 to 60 nanoseconds per random data fetch. Floating point units achieve high throughput through concurrency, allowing multiple operations to be performed simultaneously, and pipelining, starting operations before the previous operation is complete. Register files are made large enough to provide source and target registers for as many operations as can be active at one time. Main memory throughput can be enhanced by interleaving memory banks and by fetching several words simultaneously (or nearly so) from main memory. Memory performance is further enhanced by the use of caches. Two levels of caches are now typical and systems are now being designed with three levels of caches.

High performance BLAS routines typically incur significant software overhead: because to achieve near the floating point unit's peak performance, BLAS routines need an inner loop that can keep the floating point units busy, surrounded by one or more levels of blocking to keep the memory accesses in the fastest memory possible. Managing concurrency and/or pipelining requires a long inner loop which operates on several vectors at once. Each level of blocking requires additional control code and separate loops to handle portions of the matrix that are not exact multiples of the block size. For example, DGEMV[1] (double precision matrix-vector multiplication) on the PARAGON has an average software overhead of 23 microseconds (over 1000 cycles at 50 Mhz) and includes 200 instructions of error checking and case selection, 750 instructions for the transpose case and 500 for the non-transpose case[2].

3.2 BLAS execution time

The execution time for each call to a BLAS routine depends upon the hardware, the BLAS implementation, the operation requested and the state of the machine, especially the contents of the caches, at the time of the call. The time per DGEMV, or BLAS Level 2, flop is limited by the speed of the memory hierarchy level at which the matrix resides. The

[1] DGEMV performs $y = \alpha Ax + \beta y$ or $y = \alpha A^T x + \beta y$, where A is a matrix, x and y are vectors and α and β are scalars.

[2] These instruction counts include all instructions routinely executed during the main loop in reduction to tridiagonal form. Not all are executed during each call to DGEMV.

Table 3.1: BLAS execution time (Time = δ_i + number of flops $\cdot \gamma_i$ in microseconds)

		BLAS Level 3		BLAS Level 2		BLAS Level 1	
	peak flop rate	software overhead δ_3	time per flop γ_3 (Mflops/sec)	software overhead δ_2	time per flop γ_2 (Mflops/sec)	software overhead δ_1	time per flop γ_1 (Mflops/sec)
PARAGON Basic Math Library Software (Release 5.0)	50	300	.024				
(41)	87	.026					
(38)	3	.10					
(10)							
IBM SP2 ESSL 2.2.2.2	480	0	.0037				
(270)	5 .1e	0.0055					
(180)	1.2	.01					
(100)							

time per DGEMM[3], or BLAS Level 3, flop is typically limited primarily by the rate at which the floating point unit can issue and complete instructions. We will concentrate on DGEMM and DGEMV because they perform most of the flops in PDSYEVX.

Table 3.1 shows the software overhead and time per flop for the BLAS routines. These times are based on independent timings with code cached but not data cached using invocations that are typical for PDSYEVX. Recall that these parameters are used in a linear model of performance:

$$\delta + \text{number of flops} \times \gamma \qquad \text{(Line 1)}$$

In PDSYEVX we are most concerned with the time per flop for Level 3 routines and secondarily concerned with the time per flop and software overhead for Level 2 routines. For n=3840 and p=64, on the Paragon, the three largest components attributable to the items in Table 3.1 are: 28% of the PDSYEVX execution time is attributable to BLAS Level

[3]DGEMM performs $C = \alpha AB + \beta C$ or $c = \alpha A^T B + \beta C$, where A, B and C are matrices, and α and β are scalars.

3 floating point execution (not including software overhead), 8% is attributable to BLAS Level 2 floating point execution and 5% is attributable to BLAS Level 2 software overhead. (See Chapter 5 for details.) The fact that the BLAS3 software overhead for the IBM SP2 is listed as 0 stems from the fact that matrix-matrix multiply is faster for small problem sizes because they fit in cache[4].

Figure 3.1: Performance of DGEMV on the Intel PARAGON

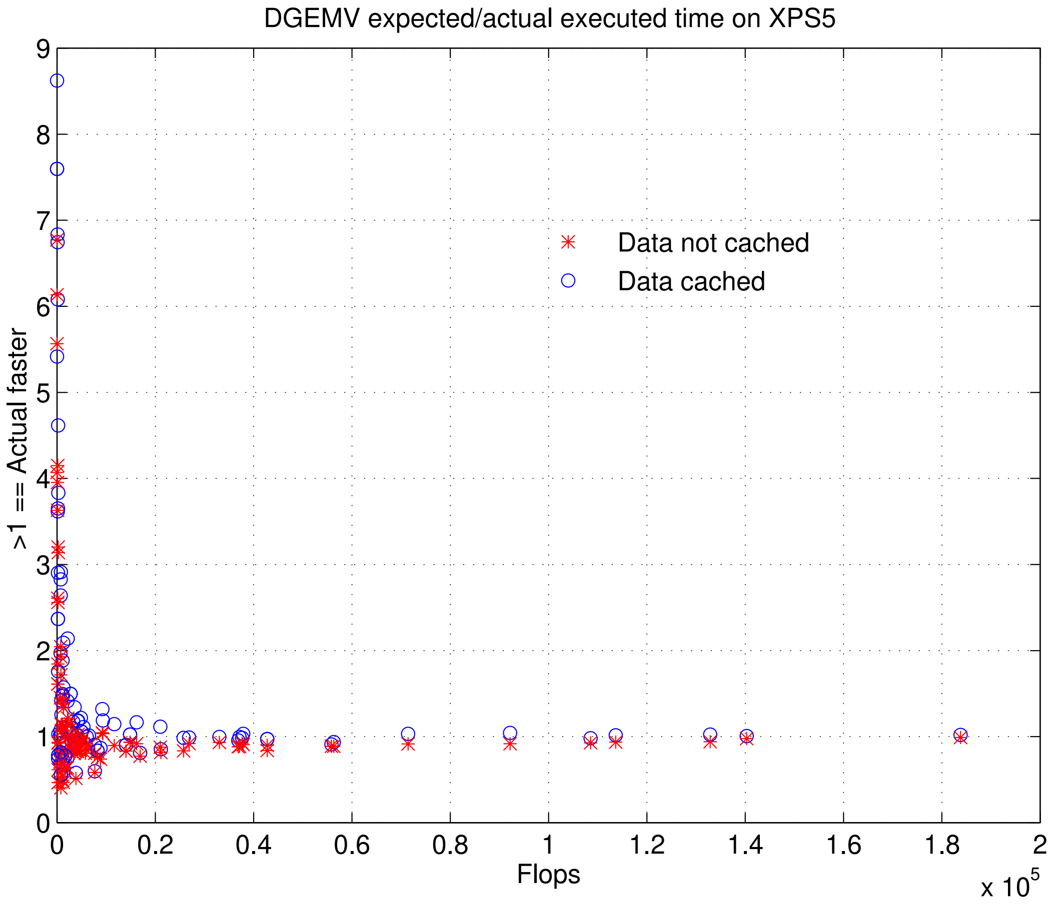


Figure 3.1 shows how actual DGEMV performance differs from predicted performance Line 1 on the PARAGON. Each point represents the time required for a call to DGEMV with parameters that are typical of calls to DGEMV made in PDSYEVX divided by the time predicted by our performance model. The timings are made by an independent timer as described in Section 3.3. The model matches quite well on most calls to DGEMV. It also shows a modest, but noticeable difference between the cost when data is cached versus when it is not. If

[4]We did not pursue this because it BLAS3 software overhead has little impact on PDSYEVX execution time.

the software overhead term were removed (i.e using number of flops $\times\gamma_2$ as the model) the model would underestimate execution by a factor of two hundred or more on small problem sizes.

Some calls to DGEMV require much less time than expected, as little as 1/9, indicating the software overhead is not independent of the type of call made. In particular, calls which involve very few flops can vary widely in their execution time (for the predicted time). However, not many calls differ widely in their execution time and those that do require few flops (hence little execution time) and the fact that they do not match well does not significantly affect the accuracy of my performance model for PDSYEVX (given in Chapter 4) and hence I did not study them further.

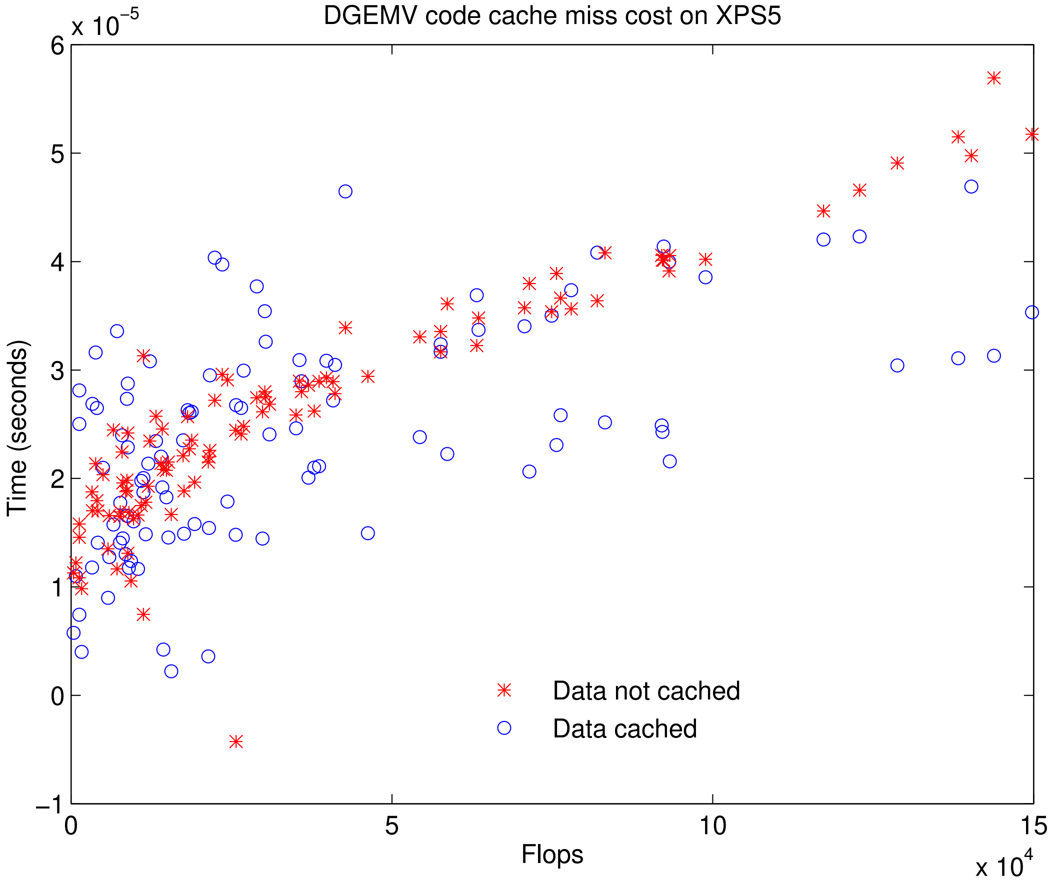
Figure 3.2 shows that DGEMV on the PARAGON requires 10 to 50 microsends longer if the code is not cached at the time it is is called. The additional time required is estimated by subtracting the cost of running DGEMV alone from the cost of running DGEMV followed by 16,384 no-ops[5] while accounting for the execution time of the 16,384 no-ops themselves. The extra time required increases as the number of flops increases. And the extra time is greater when the data is not cached than when it is cached[6]. It is not surprising that the extra time required when the code is not cached increases as the number of flops increases because when few flops are involved, the code does not execute as many loops. However it is surprising that the code cache miss cost in the "Data not cached" case appears to increase almost linearly with the number of flops, I would expect to see something closer to a step function. This deserves further study if it is determined that code cache misses substantially affect execution time.

Figure 3.3 shows that the extra time required by DGEMV ranges from 1.5% (when DGEMV performs many flops) to over 10% (when DGEMV performs few flops). Only calls made to DGEMV with parameters that are typical of the calls commonly made by PDSYEVX are shown. The extra time required when code is not cached can be up to 80% on calls made to DGEMV requiring very few flops, but these are rare in PDSYEVX.

[5]The code cache holds 8,192 no-ops. Hence, 16,384 guarantees that the no-ops are not in cache, making their execution time independent of what is in the code cache at the time the 16,384 no-ops are executed.

[6]I compare the execution time when neither code nor data is cached to the execution time when code is cached but data is not when estimating the extra time required when data is not cached.

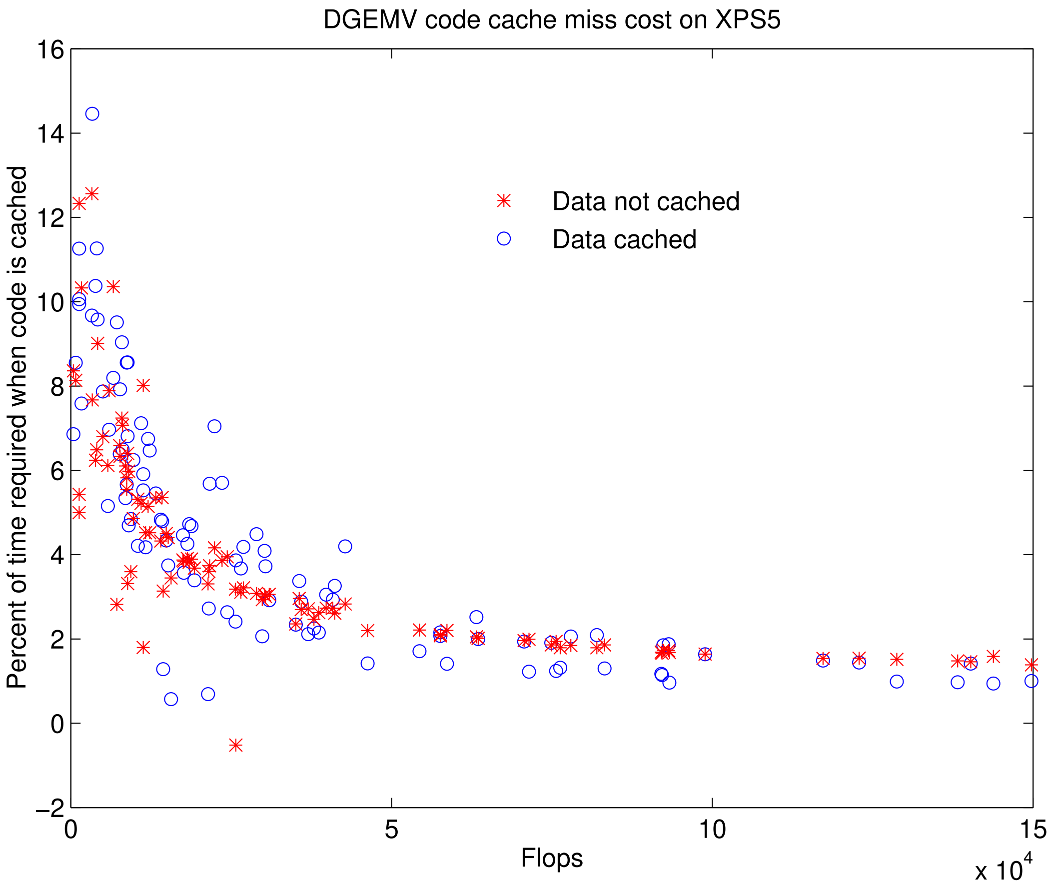
Figure 3.2: Additional execution time required for DGEMV when the code cache is flushed between each call. The y-axis shows the difference between the time required for a run which consists of one loop executing 16,384 no-ops after each call to DGEMV and the time required for a run which includes two loops one executing DGEMV and one executing 16,384 no-ops.



3.3 Timing methodology

Each routine is timed with several sets of input parameters. To time a routine with a given set of input parameters, the routine is run three times and the time from the third run is used. Each run consists of calling the routine to be timed repeatedly within a loop. The first run, in which the loop is run only once, ensures that the code is paged in. The second run, in which the loop is run just long enough to exceed the timer resolution, provides an estimate that is used to determine how many times to run the third run. The third run, in which the loop is run for approximately one second, is the only one whose execution time is recorded. We record both CPU time and wall clock time. These plots are

Figure 3.3: Additional execution time required for DGEMV when the code cache is flushed between each call as a percentage of the time required when the code is cached. See Figure 3.2.



based on CPU time.

The input parameters for each run are randomly selected such that they match the input parameters made in a typical call to DGEMV from PDSYEVX. Randomly selecting the input parameters provides advantages over a systematic choice of input parameters. A systematic choice of input parameters might include, for example, only even values of k whereas odd values of k might require significantly longer. Random selection means that the likelihood of identifying anomalous behavior is directly related to how often that behavior occurs in calls within PDSYEVX. Random selection scales well also: It is easy to increase or decrease the number of timings and/or the number of processors used.

3.4 The cost of code and data cache misses in DGEMV

Each set of input parameters is timed under four different cache situations:

Code and data cached

Code cached but data not cached

Code not cached but data cached

Neither code nor data cached

Data can be allowed to remain in cache (to the extent that it fits in cache) by using the same arrays in each call within the timing loop. Likewise data can be prevented from remaining in cache by using different arrays for each call within the timing loop.

Allowing data to reside in cache reduces execution time in two ways. It reduces the cost of accessing the data in the arrays being operated on and it reduces the software overhead cost, because software overhead also involves reading and writing data, notably while saving and restoring registers.

Code and data cache misses are more important in DGEMV than in DGEMM because DGEMV is called more often than DGEMM and the ratio of flops to data movement is higher for DGEMM than for DGEMV, hence reducing the cost of data cache misses in DGEMM.

3.5 Miscellaneous timing details

We make sure that timings are not affected by conditions which are not likely to be encountered in a typical run of PDSYEVX. Exceptional numbers (subnormalized numbers and infinities) will occur only rarely in PDSYEVX[7]. Hence, we make sure that exceptional numbers do not appear during our timing runs.

We do not time PDSYEVX on problem sizes that do not fit in physical memory. Hence, when timing the individual BLAS routines, we make sure that the arrays fit in physical memory. Ed D'Azevedo has written an out-of-core symmetric eigensolver and studied the affect of paging on PDSYEVX[52].

[7] The matrix is scaled before reduction to tridiagonal form to avoid being close to the overflow or underflow threshold. Although this does not prevent underflows (or subnormalized numbers) it causes them to be rare. NaNs will never appear in PDSYEVX unless NaNs appear in the input.

We measure and report both wall clock time and CPU time. Wall clock time may differ from CPU time for several reasons, including: time spent waiting for communication, time spent on other processes and time spent on paging and other operating system services. When timing the `BLAS`, we are primarily interested in CPU time because we there is no communication and we are not interested in measuring the time spent waiting on other processes. However, we measure and report wall clock time because for all other timings we must rely on wall clock timings[8]. When the wall clock time differs substantially from the CPU time on calls to the `BLAS` on time shared systems (such as the `IBM SP2`) we use the ratio of wall clock time to CPU time as a crude measure of the load on the system.

We use the timing routines included in the `BLACS` routines developed at `Univeristy of Tennessee at Knoxville`[169, 69] (which are not a part of the `BLACS` specification). Many modern computers have cycle time counters which would allow much more detailed measurement of execution time and often other machine characteristics. These detailed timing routines are not portable and I chose to stick to portable timing techniques. Alternatively, Krste Asanovic has developed a portable interface for taking performance related statistics over an "interval" of a code's execution[11].

[8]CPU time is often meaningless when communication is involved.

Chapter 4

Details of the execution time of PDSYEVX

4.1 High level overview of PDSYEVX algorithm

Figure 4.1 shows how PDSYEVX reduces the original (dense) matrix to tridiagonal form (Line 1), uses bisection and inverse iteration to solve the tridiagonal eigenproblem (Line 2) and then transforms the eigenvectors of the tridiagonal matrix back into the eigenvectors of the original dense matrix(Line 3). PDSYEVX uses a two-dimensional block cyclic data layout with an algorithmic block size equal to the data layout block size in both Householder reduction to tridiagonal form and back transformation. When using bisection to compute the eigenvalues, it assigns each process an essentially equal number of eigenvalues to compute. For inverse iteration, PDSYEVX attempts to assign roughly equal numbers of eigenvectors to each process while assigning all eigenvectors corresponding to a given cluster of eigenvalues to the same process. Gram-Schmidt re-orthogonalization is performed locally within each process and hence orthogonality is not guaranteed for eigenvectors corresponding to eigenvalues within a cluster that is too large to fit on a single process.

We assume that only the lower triangle of the square symmetric matrix A contains valid data on input and the algorithms only read and write this lower triangle. The general conclusions of this thesis apply to the upper triangular case as well.

Please refer to Table A.1, Table A.2, and Table A in Appendix Afor the list of

Figure 4.1: PDSYEVX algorithm

$A = QTQ^T$	$A \in R^n$ is the matrix whose eigendecomposition we seek. T is tridiagonal. Q is orthogonal.	(Line 1)
$T = U\Lambda U^T$	$\Lambda = \mathsf{diag}(\lambda_1, \ldots, \lambda_n)$ is the diagonal matrix of eigenvalues. The columns of $U = [u_1 \ldots u_n]$ are the eigenvectors of T. $Tu_i = \lambda_i u_i$	(Line 2)
$V = QU$	The columns of $V = [v_1 \ldots v_n]$ are the eigenvectors of A. $Av_i = \lambda_i v_i$	(Line 3)

notation used in this chapter.

Section 4.2 describes and models reduction to tridiagonal form as performed by PDSYTRD. Section 4.3 describes and models the tridiagonal eigensolution as performed by PDSTEBZ (bisection) and PDSTEIN(inverse iteration). Section 4.4 describes and models back transformation as performed by PDORMTR.

4.2 Reduction to tridiagonal form

4.2.1 Householder's algorithm

Figure 4.4 shows Householder's reduction to tridiagonal form, Figure 4.4 shows a model for the runtime of ScaLAPACK's reduction to tridiagonal form code, PDSYTRD. The rest of this section explains the computation and communication pattern in PDSYTRD. We begin by describing the classical (serial and unblocked) algorithm (essentially the EISPACK algorithm TRED1 and also LAPACK's DSYTD2), then the blocked (but still serial) algorithm (essentially the LAPACK algorithm DSYTRD) and finally the parallel blocked ScaLAPACK algorithm PDSYTRD.

Classical (serial and unblocked) Householder reduction (Figure 4.2)

Figure 4.2 shows the algorithm for the clasical (serial and unblocked) Householder reduction to tridiagonal form, (essentially the algorithm used in LAPACK's DSYTD2.

The first iteration through the loop performs an orthogonal similarity transformation of the form: $A \leftarrow (I - \tau v v^t) A (I - \tau v v^t)$ where $\tau = 2 / \parallel v_2 \parallel_2^2$, such that only the first two elements in the first column (and hence the first two elements in the first row) of A are non-zero. Each iteration through the loop repeats these steps on the trailing submatrix $A(2{:}n, 2{:}n)$ to reduce A to tridiagonal form by a series of similarity transformations.

Compute an appropriate reflector **(Line 2.1 in Figure 4.2)**

We seek a reflector of the form: $I - \tau v v^t$ such that $\tau = \frac{2}{v^t v}$ and the first row and column of $(I - \tau v v^t) A (I - \tau v v^t)$ has zeroes in all entries except the first two.

Let z be the column vector $A(2{:}n, 1)$. In exact arithmetic, any vector $v = c[z_1 \pm \parallel z \parallel_2, z_2 \ldots z_n]$ for any scalar c will suffice, and defines what value τ must take. LAPACK and ScaLAPACK choose the sign ($\pm \parallel z \parallel_2$) to match the sign of z_1 to minimize roundoff errors, and choose c such that $v(1) = 1.0$. c can also be chosen to be 1, avoiding the need to multiply z by c, at some small risk of over/underflow.

Form the matrix vector product $y = Av$ **(Line 3.3 in Figure 4.2)**

This is a matrix vector multiply (Basic Linear Algebra Subroutines Level 2) requiring $2(n-i)^2$ flops, which when summed from $i = 1$ to $n-1$ totals $\frac{2}{3} n^3$ flops.

Compute the companion update vector $w = y - \frac{1}{2}\left(\tau (y)^T \cdot v\right) v$ **(Line 5.1 in Figure 4.2)**

The vector w (which is computed here with a dot product and a DAXPY) has the property that $(I - \tau v v^T) A (I - \tau v v^T) = A - v w^T - w v^T$.

Update the matrix **(Line 6.3 in Figure 4.2)**

Compute $A = A - v w^T - w v^T$, a BLAS Level 2 rank-2 update. A rank-2 update requires 4 flops per element updated, only the lower triangular portion of A is updated, so this requires $2(n-i)^2$ flops, which summed over $i = 1$ to $n-1$ is $\frac{2}{3} n^3$ flops.

Figure 4.2: Classical unblocked, serial reduction to tridiagonal form, i.e. EISPACK's TRED1(The line numbers are consistent with figures 4.3, 4.4 and 4.5.)

do $i = 1, n$

Compute reflector

Line	Operation	Comment
2.1	$[\tau, v] = \text{house}(A(i{+}1{:}n, i))$	$v \in R^{n-i}$; τ is a scalar; House computes a householder vector such that $(I - \tau v v^T)A(i{+}1{:}n, i)(I - \tau v v^T)$ is zero except for the top element.

Perform matrix-vector multiply

Line	Operation	Comment
3.3	$w = \text{tril}(A(i{+}1{:}n, i{+}1{:}n))v$ $+ \text{tril}(A(i{+}1{:}n, i{+}1{:}n), -1)v^T$	$w \in R^{n-i}$; tril() is MATLAB notation for the lower triangular portion of a matrix (including the diagonal). tril$(, -1)$ refers to the portion of the matrix below the diagonal.

Compute companion update vector

5.1 $c = w \cdot v^T$;
$w = \tau\, w - (c\,\tau/2)\, v$

Perform rank 2 update

Line	Operation	Comment
6.3	$A(i{+}1{:}n, i{+}1{:}n) =$ $\text{tril}(A(i{+}1{:}n, i{+}1{:}n)$ $- w v^T - v w^T)$	Here we use tril to indicate that only the lower triangular portion of A need be updated.

end do $i = 1, n$

Blocked Householder reduction to tridiagonal form (Figure 4.3)

In the above algorithm, nearly all the flops are performed in the product $y = Av$, or the rank-2 update $A - vw^T - wv^T$, both of which are BLAS Level 2 operations. Through blocking, half of the flops can be executed as BLAS 3 flops because k matrix updates can be performed as one rank-$2k$ update instead of k rank-2 updates. This is done in Line 6.3 in Figure 4.2. The cost of blocking is significant in PDSYTRD, but the gain is also. See section 7.2.2. This allows the matrix update to be considerably more efficient, but it complicates the computation of the reflector and the computation of the companion update vector, because PDSYTRD must work with an out-of-date matrix. Starting with A_0, the computation of the first reflector v_0, the matrix vector product and w_0 are unchanged, but as soon as PDSYTRD attempts to compute the second reflector, v_1 it has to deal with the fact that A_1 is known only in factored from, i.e. $A_1 = A_0 - v_0 w_0^T - w_0 v_0^T$. This does not greatly complicate computing the reflector because the reflector needs only the first column of A_1.

Figure 4.3: Blocked, serial reduction to tridiagonal form, i.e. `DSYEVX`(See Figure 4.2 for unblocked serial code)

do $ii = 1, n, \mathsf{nb}$
  $mxi = \min(ii + \mathsf{nb}, n)$
  do $i = ii, mxi$

Update current (i^{th}) column of A

$A(:, i) = A(:, i) -$
1.2 $W(:, ii{:}i{-}1)\, V(i, ii{:}i{-}1)^T -$
$V(:, ii{:}i{-}1)\, W(i, ii{:}i{-}1)^T$

Compute reflector

2.1 $[\tau, v] = \mathrm{house}(A(i{+}1{:}n, i))$ — $v \in R^{n-i}$; τ is a scalar

Perform matrix-vector multiply

3.3 $w = \mathrm{tril}(A(i{+}1{:}n, i{+}1{:}n))v$ — $w \in R^{n-i}$
$+ \, \mathrm{tril}(A(i{+}1{:}n, i{+}1{:}n), -1)^T v$

Update the matrix-vector product

$w = w -$
4.1 $W(:, ii{:}i{-}1)\, V(i, i{+}1{:}n)^T v -$
$V(:, ii{:}i{-}1)\, W(i, i{+}1{:}n)^T v$

Compute companion update vector

5.1 $c = w \cdot v^T$;
$w = \tau\, w - (c\, \tau/2)\, v$

$W(i{+}1{:}n, i) = w$;
$V(i{+}1{:}n, i) = v$
end do $i = ii, mxi$

Perform rank $2k$ update

$A(mxi{+}1{:}n, mxi{+}1{:}n) =$
$\mathrm{tril}(A(mxi{+}1{:}n, mxi{+}1{:}n) -$
6.3 $W(mxi{+}1{:}n, ii{:}mxi)V(mxi{+}1{:}n, ii{:}mxi)^T -$
$V(mxi{+}1{:}n, ii{:}mxi)W(mxi{+}1{:}n, ii{:}mxi)^T)$
end do $ii = 1, n, \mathsf{nb}$

However, computing w_1 requires the computation of $A_1 v$, hence we must either update the entire matrix A_1, returning to an unblocked code, or compute $y = (A_0 - v_0 w_0^T - w_0 v_0^T)v$. Computing the reflectors and the companion update vectors now requires that the current column be updated (Line 1.2 in Figure 4.3). The matrix vector product must be updated (Line 4.1 in Figure 4.3).

4.2.2 PDSYTRD implementation (Figure 4.4)

Figure 4.5 shows Householder's reduction to tridiagonal form along with a model for the runtime of each step in ScaLAPACK's reduction to tridiagonal form code, PDSYTRD. The rest of this section explains the computation and communication pattern in PDSYTRD, and hence the inefficiencies.

Figure 4.4: PDSYEVX reduction to tridiagonal form (See Figure 4.3 for further details)

do $ii = 1, n, \mathsf{nb}$
  $mxi = \min(ii + \mathsf{nb}, n)$
  do $i = ii, mxi$

Update current (i^{th}) column of A (Table 4.1)

1.1	spread $V(i, ii{:}i{-}1)^T$ and $W(i, ii{:}i{-}1)^T$ down	processor owning $V(i, ii : i{-}1)$ and $W(i, ii{:}i{-}1)$ broadcasts to all other processors in its procesor column.
1.2	$A(i,i) = A(:,i) - W(:, ii{:}i{-}1)\, V(i, ii{:}i{-}1)^T - V(:, ii{:}i{-}1)\, W(i, ii{:}i{-}1)^T$	V and W are used as they are stored (no data movement required)

Compute reflector (Table 4.2)

2.1	$[\tau, v] = \text{house}(A(i{+}1{:}n, i))$	$v \in R^{n-i}$; τ is a scalar

Perform matrix-vector multiply (Table 4.3)

3.1	spread v across	
3.2	transpose v, spread down	
3.3	$w_1 = \text{tril}(A(i{+}1{:}n, i{+}1{:}n))v$;	w_1 is distributed like row $A(i,:)$
	$w_2 = \text{tril}(A(i{+}1{:}n, i{+}1{:}n), -1)^T v$	w_2 is distributed like column $A(:,i)$
3.4	sum w row-wise	
3.5	sum w^T column-wise	
3.6	$w = w_1 + w_2$	w is distributed like column $A(:,i), hence w_1$ must be transposed.

Update the matrix-vector product (Table 4.4)

4.1	$w = w - W(:, ii{:}i{-}1)\, V(i, i{+}1{:}n)^T v - V(:, ii{:}i{-}1)\, W(i, i{+}1{:}n)^T v$

Compute companion update vector (Table 4.5)

5.1	$c = w \cdot v^T$;
	$w = \tau\, w - (c\, \tau/2)\, v$

  $W(i{+}1{:}n, i) = w$;
  $V(i{+}1{:}n, i) = v$
  end do $i = ii, mxi$

Perform rank $2k$ update (Table 4.6)

6.1	spread $V(mxi{+}1{:}n, ii{:}mxi)$, $W(mxi{+}1{:}n, ii{:}mxi)$ across	processors in current column of processors broadcasts to processors in other processor columns
6.2	transpose $V(mxi{+}1{:}n, ii{:}mxi)$, $W(mxi{+}1{:}n, ii{:}mxi)$, spread down	
6.3	$A(mxi{+}1{:}n, mxi{+}1{:}n) = \text{tril}(A(mxi{+}1{:}n, mxi{+}1{:}n) - W(mxi{+}1{:}n, ii{:}mxi)V(mxi{+}1{:}n, ii{:}mxi)^T - V(mxi{+}1{:}n, ii{:}mxi)W(mxi{+}1{:}n, ii{:}mxi)^T)$	

end do $ii = 1, n, \mathsf{nb}$

Figure 4.5: Execution time model for `PDSYEVX` reduction to tridiagonal form (See Figure 4.4 for details about the algorithm and indices.)

		computation		communication	
		overhead	imbalance	latency	bandwidth
	do $ii = 1, n, \mathsf{nb}$				
	$mxi = \min(ii + \mathsf{nb}, n)$				
	do $i = ii, mxi$				
Update current (i^{th}) column of A					
1.1	spread V^T and W^T down			$2\,n\,\mathsf{lg}(\sqrt{p})\,\alpha$	
1.2	$A = A - W\,V^T - V\,W^T$	$2\,n\,\delta_4$	$\frac{n^2\,\mathsf{nb}}{\sqrt{p}}\,\gamma_2$	$2\,n\,\mathsf{lg}(\sqrt{p})\,\alpha$	
Compute reflector					
2.1	$v = \mathrm{house}(A)$	$n\,\delta_4$		$3\,n\,\mathsf{lg}(\sqrt{p})\,\alpha$	
Perform matrix-vector multiply					
3.1	spread v across			$n\,\mathsf{lg}(\sqrt{p})\,\alpha$	$\frac{1}{2}\,\frac{n^2\,\mathsf{lg}(\sqrt{p})}{\sqrt{p}}\,\beta$
3.2	transpose v, spread down	$\frac{n^2}{\sqrt{p}}\,\delta_1$		$n\,\mathsf{lg}(\sqrt{p})\,\alpha$	$\frac{1}{2}\,\frac{n^2\,\mathsf{lg}(\sqrt{p})}{\sqrt{p}}\,\beta$
3.3	$w = \mathrm{tril}(A)v$; $w^T = \mathrm{tril}(A,-1)v^T$	$(n\delta_4 + \frac{n^2}{\mathsf{nb}\sqrt{p}}\,\delta_2)$	$(\frac{2}{3}\frac{n^3}{p}\gamma_2 + 3\,\frac{n^2\,\mathsf{nb}}{\sqrt{p}}\gamma_2)$		
3.4	sum w row-wise			$n\,\mathsf{lg}(\sqrt{p})\,\alpha$	$\frac{1}{2}\,\frac{n^2\,\mathsf{lg}(\sqrt{p})}{\sqrt{p}}\,\beta$
3.5	sum w^T column-wise			$n\,\mathsf{lg}(\sqrt{p})\,\alpha$	$\frac{1}{2}\,\frac{n^2\,\mathsf{lg}(\sqrt{p})}{\sqrt{p}}\,\beta$
3.6	$w = w + \text{transpose } w^T$				
Update the matrix-vector product					
4.1	$w = w - W\,V^T v - V\,W^T v$	$4\,n\,\delta_4$	$2\,\frac{n^2\,\mathsf{nb}}{\sqrt{p}}\,\gamma_2$	$6\,n\,\mathsf{lg}(\sqrt{p})\,\alpha$	$\frac{n^2\,\mathsf{lg}(\sqrt{p})}{\sqrt{p}}\,\beta$
Compute companion update vector					
5.1	$c = w \cdot v^T$; $w = \tau\,w - (c\,\tau/2)\,v$	$n\,\delta_4$		$2n\,\mathsf{lg}(\sqrt{p})\,\alpha$	
	end do $i = ii, mxi$				
Perform rank $2k$ update					
6.1	spread V, W across				$\frac{n^2\,\mathsf{lg}(\sqrt{p})}{\sqrt{p}}\,\beta$
6.2	transpose V, W, spread down				$\frac{n^2\,\mathsf{lg}(\sqrt{p})}{\sqrt{p}}\,\beta$
6.3	$A = A - W\,V^T - V\,W^T$	$2\,\frac{n^2}{\mathsf{nb}^2\sqrt{p}}\,\delta_3$	$(\frac{2}{3}\frac{n^3}{p}\gamma_3 + 3\,\frac{n^2\,\mathsf{nb}}{\sqrt{p}}\,\gamma_3)$		
	end do $ii = 1, n, \mathsf{nb}$				

Distribution of data and computation in PDSYTRD

In PDSYEVX, the matrix being reduced, A, is distributed across a 2 dimensional grid of processors. The computation is distributed in a like manner, i.e. computations involving matrix element $A(i,j)$ are performed by the processor which owns matrix element $A(i,j)$. Vectors are distributed across the processors within a given column of processors. At the i^{th} step, i.e. when reducing $A(i:n,i:n)$ to $A(i{+}1:n,i{+}1:n)$, the vectors are distributed amongst the processors which own some portion of the vector $A(i:n,i)$. Within calls to the PBLAS, these vectors are sometimes replicated across all processor columns, or even transposed and replicated across all processor rows. However, between PBLAS calls, each vector element is owned by just one processor.

Critical path in PDSYTRD

For steps 1.1, 1.2, 2.1, 4.1, 5.1, 6.1, 6.2, 6.3 in Figure 4.5, i.e. all steps except "forming the matrix vector product", the processor owning the most rows in the current column of the remaining matrix has the most work to do and hence it is on the critical path. When the matrix vector product is being formed, (steps 3.1 through 3.6) the processor which owns the most rows and the most columns in the remaining matrix has the most work (both communication and computation) and hence is on the critical path.

Load imbalance

Load imbalance occurs when some processor(s) take longer to perform certain operations[1], requiring other processors to wait. Each processor is responsible for computations on the portion of the matrix and/or vectors that it owns. Some processors own a larger portion of the matrix and/or vectors. Since PDSYTRD has regular synchronization points[2], the processor which takes the longest to complete any given step determines the execution time for that step.

If row j is the first row in a data layout block, the processor which owns $A(j,j)$ will own the most rows in $A(j:n,j:n)$: $\lfloor \frac{n-j+1}{p_r\,\mathsf{nb}} \rfloor\, \mathsf{nb} + \min(n-j+1-\lfloor \frac{n-j}{p_r\,\mathsf{nb}} \rfloor\, \mathsf{nb}\, p_r, \mathsf{nb})$. However, if row j is not the first row in a data layout block, even this formula is too simplistic.

[1]Load imbalance also occurs during communication, but for PDSYTRD on the machines that we studied the communicaiton load imbalance was negligible.

[2]Computing the reflector (Line 2.1) and computing the companion update vector (Line 5.1) require all the processors in the processor column owning column i of the matrix and are hence synchronization points.

Fortunately, $\frac{n-j+1}{p_r}+\frac{\mathtt{nb}}{2}$ is an excellent approximation, on average, for the maximum number of rows of $A(j\!:\!n, j\!:\!n)$ owned by any processor. $\frac{n-j+1}{p_r}+\frac{\mathtt{nb}}{2}\frac{p_r-1}{p_r}$ is more accurate, but the difference is too small to be useful.

The second source of load imbalance is that many of the computations are performed only by the processors which own the current column of the matrix.

Updating the current column of A

As shown in table 4.1, `PDSYTRD` updates the current column of A through two calls to `PDGEMV`, one at line 350 of *pdlatrd.f* and one at line 355 of *pdlatrd.f*. Each of these calls to `PDGEMV` requires that the first few elements of a column vector (W or V) be transposed and replicated among all the processors in that column. The transposition is fast because these elements are entirely contained within one processor, but the replication requires a spread down (column-wise broadcast) of `nb` or fewer items.

Standard data layout model

By making a few assumptions, we can significantly simplify the model. By assuming that $p_r = p_c = \sqrt{p}$, many of the terms coalesce. We also assume that the panel blocking factor[4] , `pbf`,= 2, as it is in `ScaLAPACK` 1.5.

This standard data layout is also assumed in Figure 4.5 and in Chapter 5. The models used in Figure 4.5 and in Chapter 5 are subsets, including only the most important terms, of the "standard data layout" models shown in Tables 4.2 through 4.10.

Computing the reflector (Line 2.1 in Figure 4.5)

`PDLARFG` computes the reflector as shown in table 4.2. First, it broadcasts $\alpha = A(j+1, j)$ to all processes that own column A(:,j). Then, it computes the norm $\beta = |A(j+1\!:\!n, j)|$ leaving the result replicated across all processors that own column $A(:, j)$.

The rest of the computation is entirely local and requires only $\frac{2n^2}{\sqrt{p}} + O(n)$ flops, hence does not contribute significantly to total execution time.

[4]The matrix vector multiplies are each performed in panels of size `pbfnb`. See Section 4.2.2.

Table 4.1: The cost of updating the current column of A in `PDLATRD`(Line 1.1 and 1.2 in Figure 4.5)

Task	File:line number or subroutine	Execution time contribution from columns $j = 1$ to n shown explicitly	Execution time (simplified)
Broadcast $W(j,1{:}j'-1)^T$ within current column[3].	*pdlatrd.f*:350 *pdgemv_.c* *pbdgemv.f*:560 *dgebs2d*	$\sum_{j=1}^{n} \left(\lceil \log_2(p_r) \rceil \alpha + \delta_4 + j' \lceil \log_2(p_r) \rceil \beta \right)$	$n \lceil \log_2(p_r) \rceil \alpha + n\,\delta_4 + 0.5 n\,\mathsf{nb}\,\lceil \log_2(p_r) \rceil \beta$
Compute local portion of $A(j{:}n) = A(j{:}n,j) - V(j{:}n,1{:}j'-1) \times W(j,1{:}j'-1)^T$	*pdlatrd.f*:350 *pdgemv_.c* *pbdgemv.f*:580 *dgemv*	$\sum_{j=1}^{n} \left(\delta_2 + 2\,\frac{(n-j)\,j'}{p_r}\,\gamma_2 \right)$	$n\,\delta_2 + 0.5\,\frac{n^2\,\mathsf{nb}}{p_r}\gamma_2$
Broadcast $V(j,1{:}j'-1)^T$ within current column.	*pdlatrd.f*:355 *pdgemv_.c* *pbdgemv.f*:560	$\sum_{j=1}^{n} \left(\lceil \log_2(p_r) \rceil \alpha + \delta_4 \; j' \lceil \log_2(p_r) \rceil \beta \right)$	$n \lceil \log_2(p_r) \rceil \alpha + n\,\delta_4 + 0.5\,n\,\mathsf{nb}\,\lceil \log_2(p_r) \rceil \beta$
Compute local portion of $A(j{:}n) = A(j{:}n,j) - W(j{:}n,1{:}j'-1) \times V(j,1{:}j'-1)^T$	*pdlatrd.f*:355 *pdgemv_.c* *pbdgemv.f*:580 *dgemv*	$\sum_{j=1}^{n} \left(\delta_2 + 2\frac{(n-j)\,j'}{p_r}\gamma_2 \right)$	$n\,\delta_2 0.5\,\frac{n^2\,\mathsf{nb}}{p_r}\gamma_2$
Total	$2\,n \lceil \log_2(p_r) \rceil \alpha + n\,\mathsf{nb}\,\lceil \log_2(p_r) \rceil \beta + 2\,n\,\delta_2 + \frac{n^2\,\mathsf{nb}}{p_r}\,\gamma_2 + 2\,n\,\delta_4$		
Standard data layout (See section 4.2.2)	$2\,n \lceil \log_2(\sqrt{p}) \rceil \alpha + n\,\mathsf{nb}\,\lceil \log_2(\sqrt{p}) \rceil \beta + 2\,n\,\delta_2 + \frac{n^2\,\mathsf{nb}}{\sqrt{p}}\,\gamma_2 + 2\,n\,\delta_4$		

Table 4.2: The cost of computing the reflector (`PDLARFG`) (Line 2.1 in Figure 4.5)

Task	File:line number or subroutine	Execution time contribution from columns $j = 1$ to n shown explicitly	Execution time (simplified)
$\alpha = A(j+1, j)$	*pdlatrd.f*:364 *pdlarfg.f*:213 *dgebs2d*	$\sum_{j=1}^{n} \lceil \log_2(p_r) \rceil \, \alpha$	$n \lceil \log_2(p_r) \rceil \, \alpha$
xnorm $= \lvert A(j+1{:}n, j) \rvert$	*pdlatrd.f*:364 *pdlarfg.f*:229 *pdnrm2*	$\sum_{j=1}^{n} \left(2 \lceil \log_2(p_r) \rceil \, \alpha + \frac{1}{2} \delta_4 \right)$	$2\, n \lceil \log_2(p_r) \rceil \, \alpha + \frac{n}{2} \delta_4$
$\tau = -(\alpha + \beta)/\beta$	*pdlatrd.f*:364 *pdlarfg.f*:271	negligible	negligible
$A(j+2, j) = \frac{A(j+2{:}n,j)}{(\alpha+\beta)}$	*pdlatrd.f*:364 *pdlarfg.f*:272 *pdscal*	$\sum_{j=1}^{n} \frac{1}{2} \delta_4$	$\frac{n}{2} \delta_4$
$E(j) = A(j+1, j) = \beta$	*pdlatrd.f*:364 *pdlarfg.f*:273	negligible	negligible
Total	$3n \lceil \log_2(p_r) \rceil \, \alpha + n \, \delta_4$		
Standard data layout (See section 4.2.2)	$3n \lceil \log_2(\sqrt{p}) \rceil \, \alpha + n \, \delta_4$		

Forming the matrix vector product using PDSYMV(Lines 3.1 through 3.6 in Figure 4.5)

The matrix A is laid out in a block cyclic manner as described in section 2.5.4. Computing the matrix vector product $y = Av$ requires that v be copied to all processes that own a part of A that needs to be multiplied by v. The vector v must be transposed[5]. Each element is sent directly from the processor (in the processor column) Each processor in the processor column that owns v sends to each processor in the processor row v exactly the elements and spread down and because only half of A is stored, v must also be spread across. Then, the matrix vector multiplies[6], $w_1 = \text{tril}(A,0)v$ and $w_2 = \text{tril}(A,-1)^T v$ are performed locally. w_1 is summed within columns, transposed and added to the result of $\text{tril}(A,0)^T v$ which is summed to the active column of processors. The algorithm used by PDSYMV is:

Algorithm 4.1 *PDSYMV as used to compute Av*

1 *Broadcast v within each row of processors (Line 3.1 in Figure 4.4)*
2 *Transpose v within each column of processors (Line 3.2 in Figure 4.4)*
3 *Broadcast v^T within each column of processors (Line 3.2 in Figure 4.4)*
4 *Form diagonal portion of A (Line 3.3 in Figure 4.4)*
5 $w_1 =$ *locally available portion of* $\text{tril}(A,0)v$ *(Line 3.3 in Figure 4.4)*
6 $w_2 = v^T\,\text{tril}(A,-1)$ *(Line 3.3 in Figure 4.4)*
7 *Sum w_1 within each column of processors (Line 3.4 in Figure 4.4)*
8 *Sum w_2 within each row of processors (Line 3.5 in Figure 4.4)*
9 *Transpose w_1 and add to w_2 (Line 3.6 in Figure 4.4)*

The two transpose operations, steps {2,3} and step 9 in algorithm 4.1 though both are performed by PBDTRNV, use different communication patterns. The transpose performed in steps 2 and 3, is an all-to-all. It takes v replicated across the processor columns and distributed across the processor rows and produces v^T replicated across the processor rows and distributed across the processor columns. The transpose performed in step 9 is a one-to-one transpose. It takes y_u^T distributed across the processor columns within one processor

[5]The non-transposed v is distributed like column $A(:,i)$, the transposed v is distributed like row $A(i,:)$.

[6]tril() is MATLAB notation for the lower triangular portion of a matrix (including the diagonal). $\text{tril}(,-1)$ refers to the portion of the matrix below the diagonal.

row. It produces y_u distributed across the processor rows within the current processor column.

The all-to-all transposition is performed in two steps (steps 2 and 3 in algorithm 4.1). Since each column of processors contains a complete copy of the vector v, each acts independently, first collecting the portion of v^T that belongs to this processor column to one processor[7] and then broadcasting it to all processor columns. The operation of collecting the portion of v^T that belongs to this processor column to one processor is done as a tree-based reduction, requiring $\lceil \log_2(\mathrm{lcm}(p_r, p_c)) \rceil$ messages, and a total of $\frac{\mathrm{lcm}(p_r,p_c)-1}{\mathrm{lcm}(p_r,p_c)} \frac{j}{p_c}$ words which I model as $\frac{n}{p_c}$ words. The broadcast which completes the transpose (step 3), requires $\lceil \log_2(p_c) \rceil$ messages and $\lceil \log_2(p_c) \rceil \frac{j}{p_c}$ words.

The one-to-one transpose (step 9) is accomplished as a single set of direct messages. Every word in y_u^T is owned by exactly one processor. Every word in y_u should be sent to one processor. Every word in y_u^T is sent from the processor that owns it to the processor that needs the corresponding word in y_u. All words being sent between the same two processors are sent in a single message. The number of words sent by each processor that owns a part of y_u^T sends every word that it owns, i.e. $\frac{j}{p_c}$ in $\mathrm{lcm}(p_r, p_c)/p_c$ messages. Every processor that needs a part of y_u receives the number of words that it needs: $\frac{j}{p_r}$ in $\mathrm{lcm}(p_r, p_c)$ messages.

The two matrix vectors multiplies are each performed in panels of size: $\mathsf{pbf}\,\mathsf{nb}$. pbf, the panel blocking factor, is set to $\max(\mathsf{mullen}, \mathrm{lcm}(p_r, p_c)/p_c)$, where mullen is a tuning parameter set at compile-time to 2 in `ScaLAPACK` 1.5.

The cost of the matrix vector multiply is detailed in table 4.3.

The number of flops in the matrix vector multiply which any given processor must perform is controlled by the size and shape of the local portion of the trailing matrix. The processor holding the largest portion of the trailing matrix holds a matrix of size approximately[8] $\lceil \frac{n-j}{\mathsf{mb}\, p_r} \rceil\, \mathsf{mb} \times \lceil \frac{n-j}{\mathsf{nb}\, p_c} \rceil\, \mathsf{nb}$. Because we update only the lower triangular portion of the matrix, each element in the lower triangular portion of the matrix is used in two matrix vector multiplies. And, because the shape of the local portion of the matrix is irregular (a column block stair step with some diagonal steps) the matrix vector computation is performed by column blocks. The irregular patterns repeats every $\frac{\mathrm{lcm}(p_r,p_c)}{p_c}\,\mathsf{nb}$, so pbf, the panel blocking factor is chosen to be: $\max(\mathsf{mullen}, \frac{\mathrm{lcm}(p_r,p_c)}{p_c})$, where mullen is a compile time

[7] If $p_r = \mathrm{lcm}(p_c, p_r)$ the portion of data that belongs othis processor column is already on one processor and hence this "collection" is a null operation.

[8] The largest local matrix size differs from this only when $\mod(n-j, \mathsf{nb}\, p_r) < \mathsf{nb}$ or $\mod(n-j, \mathsf{nb}\, p_c) < \mathsf{nb}$.

Table 4.3: The cost of all calls to `PDSYMV` from `PDSYTRD`

Broadcast v within each processor row (Line 3.1)	*pdlatrd.f*:370 *pdsymv_.c* *pbdsymv.f*:406 *dgebr2d*	$\sum_{j=1}^{n}\big(\lceil\log_2(p_c)\rceil\,\alpha+$ $\lceil\log_2(p_c)\rceil(\frac{(n-j)}{p_r}+\frac{\mathsf{nb}}{2})\beta\big)$	$n\,\lceil\log_2(p_c)\rceil\alpha+$ $0.5\frac{n^2\,\lceil\log_2(p_c)\rceil}{p_r}\beta+$ $0.5\,n\,\mathsf{nb}\,\lceil\log_2(p_c)\rceil\beta$
Transpose v (Line 3.2)	*pdlatrd.f*:370 *pdsymv_.c* *pbdsymv.f*:421 *pbdtrnv.f*:385 *pbdtrget*	$\sum_{j=1}^{n}\big(\lceil\log_2(\frac{\mathrm{lcm}(p_r,p_c)}{p_c})\rceil\alpha$ $+\frac{n-j}{p_c}\beta\big)$	$n\lceil\log_2(\mathrm{lcm}(p_r,p_c))\rceil\alpha$ $+0.5\frac{n^2}{p_c}\beta$
Broadcast v^T down within each processor column (Line 3.2)	*pdlatrd.f*:370 *pdsymv_.c* *pbdsymv.f*:421 *pbdtrnv.f*:400 *dgebs2d*	$\sum_{j=1}^{n}\big(\lceil\log_2(p_r)\rceil\,\alpha+$ $\lceil\log_2(p_r)\rceil(\frac{n-j}{p_c}+\frac{\mathsf{nb}}{2})\beta\big)$	$n\,\lceil\log_2(p_r)\rceil\alpha+$ $0.5\frac{n^2\,\lceil\log_2(p_r)\rceil}{p_c}\beta+$ $0.5n\,\mathsf{nb}\,\lceil\log_2(p_r)\rceil\beta$
Form diagonal portion of matrix, padded with zeroes (Line 3.3)	*pdlatrd.f*:370 *pdsymv_.c* *pbdsymv.f*:685 *pbdlacp1*	$\sum_{j=1}^{n}\big(\frac{n-j}{p_c}\delta_1\,\frac{n-j}{p_c}\delta_1\big)$	$\frac{1}{2}\frac{n^2}{p_c}\delta_1+\frac{1}{2}\frac{n^2}{p_c}\delta_1$
$w=\mathrm{tril}(A,0)v$, $w^T=v^T\ \mathrm{tril}(A,-1)$ local computation (Line 3.3)	*pdlatrd.f*:370 *pdsymv_.c* *pbdsymv.f*:702, 704,757,759 *dgemv*	$2\ \sum_{j=1}^{n}\big(2\frac{n-j}{\mathsf{pbf}\,\mathsf{nb}\,p_c}\delta_2+$ $(\lceil\frac{n-j}{\mathsf{nb}\,p_r}\rceil\mathsf{nb})\,(\lceil\frac{n-j}{\mathsf{nb}\,p_c}\rceil\mathsf{nb})\gamma_2$ $+j\,\frac{\mathsf{pbf}\,\mathsf{nb}}{p_r}\gamma_2\big)$	$2\frac{n^2}{\mathsf{pbf}\,\mathsf{nb}\,p_c}\delta_2+$ $\frac{2}{3}\frac{n^3}{p}\,\gamma_2+\frac{1}{2}\,\frac{n^2\,\mathsf{nb}}{p_r}\,\gamma_2+$ $\frac{1}{2}\,\frac{n^2\,\mathsf{nb}}{p_c}\,\gamma_2+\frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{p_r}\,\gamma_2$
Sum, row-wise, w (Line 3.4)	*pdlatrd.f*:364 *pdsymv_.c* *pbdsymv.f*:801 *dgsum2d*	$\sum_{j=1}^{n}\big(\lceil\log_2(p_c)\rceil\,\alpha+$ $\lceil\log_2(p_c)\rceil(\frac{(n-j)}{p_r}+\frac{\mathsf{nb}}{2})\beta\big)$	$n\,\lceil\log_2(p_c)\rceil\alpha+$ $0.5\frac{n^2\,\lceil\log_2(p_c)\rceil}{p_r}\beta+$ $0.5n\,\mathsf{nb}\,\lceil\log_2(p_c)\rceil\beta$
Sum, columnwise, w^T (Line 3.5)	*pdlatrd.f*:364 *pdsymv_.c* *pbdsymv.f*:809 *dgsum2d*	$\sum_{j=1}^{n}\big(\lceil\log_2(p_r)\rceil\,\alpha+$ $\lceil\log_2(p_r)\rceil(\frac{n-j}{p_c}+\frac{\mathsf{nb}}{2})\beta\big)$	$n\,\lceil\log_2(p_r)\rceil\alpha+$ $0.5\frac{n^2\,\lceil\log_2(p_r)\rceil}{p_c}\beta+$ $0.5\,n\,\mathsf{nb}\,\lceil\log_2(p_r)\rceil\,\beta$
Transpose w^T and sum into w (Line 3.6)	*pdlatrd.f*:370 *pdsymv_.c* *pbdsymv.f*:811 *pbdtrnv*	$\sum_{i=1}\ n\big(\frac{\mathrm{lcm}(p_r,p_c)}{p_r}\alpha+$ $\frac{\mathrm{lcm}(p_r,p_c)}{p_c}\,\alpha+\frac{n}{p_c}\beta+\frac{n}{p_r}\beta+$ $\frac{n}{\mathsf{nb}\,p_c}\delta_1+\frac{n}{\mathsf{nb}\,p_r}\delta_1\big)$	$\frac{n\ \mathrm{lcm}(p_r,p_c)}{p_r}\alpha+$ $\frac{n\ \mathrm{lcm}(p_r,p_c)}{p_c}\alpha+$ $0.5\frac{n^2}{p_c}\beta+0.5\frac{n^2}{p_r}\beta+$ $0.5\frac{n^2}{\mathsf{nb}\,p_c}\delta_1+0.5\frac{n^2}{\mathsf{nb}\,p_r}\delta_1$
Total	$2\,n\,\lceil\log_2(p_c)\rceil\,\alpha+2n\,\lceil\log_2(p_r)\rceil\alpha+n\frac{\mathrm{lcm}(p_r,p_c)}{p_r}\alpha+n\,\frac{\mathrm{lcm}(p_r,p_c)}{p_c}\alpha+$ $n\,\lceil\log_2(\mathrm{lcm}(p_r,p_c))\rceil\alpha+\frac{n^2\,\lceil\log_2(p_c)\rceil}{p_r}\beta+\frac{n^2\,\lceil\log_2(p_r)\rceil}{p_c}\beta+\frac{n^2}{p_c}\beta+0.5\frac{n^2}{p_r}\beta+$ $n\,\mathsf{nb}\,\lceil\log_2(p_c)\rceil\beta+n\,\mathsf{nb}\,\lceil\log_2(p_r)\rceil\beta+2\frac{n^2}{\mathsf{pbf}\,\mathsf{nb}\,p_c}\delta_2+$ $\frac{2}{3}\,\frac{n^3}{p}\,\gamma_2+\frac{1}{2}\,\frac{n^2\,\mathsf{nb}}{p_r}\,\gamma_2+\frac{1}{2}\,\frac{n^2\,\mathsf{nb}}{p_c}\,\gamma_2+\frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{p_r}\,\gamma_2+\frac{n^2}{p_c}\delta_1+n\,\delta_4$		
Standard data layout (See section 4.2.2)	$4n\,\lceil\log_2(\sqrt{p})\rceil\alpha+2n\alpha+2\frac{n^2\,\lceil\log_2(\sqrt{p})\rceil}{\sqrt{p}}\beta+$ $1.5\frac{n^2}{\sqrt{p}}\beta+2n\,\mathsf{nb}\,\lceil\log_2(\sqrt{p})\rceil\beta+\frac{n^2}{\mathsf{nb}\,\sqrt{p}}\delta_2+$ $\frac{2}{3}\,\frac{n^3}{p}\,\gamma_2+3\frac{n^2\,\mathsf{nb}}{\sqrt{p}}\,\gamma_2+\frac{n^2}{\sqrt{p}}\delta_1+n\,\delta_4$		

parameter, set to 2 in the standard PBLAS release. The column panels are filled out with zeroes to make the matrix vector multiply efficient. Even the act of filling the diagonal blocks with zeroes, because it is done inefficiently, is noticeable on modest problem sizes.

The number of flops required for a global $(n-j)\times(n-j)$ matrix vector multiply is approximately:

$$2 \times 2 \times \left(\frac{1}{2}\left(\left\lceil\frac{n-j}{\mathsf{nb}\,p_r}\right\rceil \mathsf{nb}\right)\left(\left\lceil\frac{n-j}{\mathsf{nb}\,p_c}\right\rceil \mathsf{nb}\right) + (n-j)\frac{\mathsf{pbf}\,\mathsf{nb}}{2\,p_r}\right).$$

The first 2 is because multiplies and adds are counted separately. Each element in the lower triangular portion of the matrix is involved twice, hence the second 2. The first term stems directly from the size of the local matrix. The second term stems from the odd shape of the local matrix and is primarily the result of the unnecessary flops (zero matrix elements) added to reduce the number of dgemv calls.

We use the following equality, dropping the $O(n)$ term:

$$\sum_{i=1}^{n}\left\lceil\frac{i}{a}\right\rceil\left\lceil\frac{i}{b}\right\rceil = \frac{n^3}{3} + \frac{n^2\,a}{4} + \frac{n^2\,b}{4} + O(n)\,.$$

$$\begin{aligned}
flops &= 2\times 2\sum_{j=1}^{n}\left(\frac{1}{2}\left\lceil\frac{n-j}{\mathsf{nb}\,p_r}\right\rceil \mathsf{nb}\,\left\lceil\frac{n-j}{\mathsf{nb}\,p_c}\right\rceil \mathsf{nb} + j\frac{\mathsf{pbf}\,\mathsf{nb}}{2\,p_r}\right)\\
&= 2\times 2\,\frac{1}{2}\left(\frac{n^3}{3\,p_r\,p_c} + \frac{1}{4}\frac{n^2}{\mathsf{nb}\,p_c}\mathsf{nb}^2 + \frac{1}{4}\frac{n^2}{\mathsf{nb}\,p_r}\mathsf{nb}^2 + \frac{n^2}{2}\frac{\mathsf{pbf}\,\mathsf{nb}}{p_r}\right)\\
&= \frac{2}{3}\frac{n^3}{p} + \frac{n^2\,\mathsf{nb}}{2p_r} + \frac{n^2\,\mathsf{nb}}{2p_c} + \frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{p_r}
\end{aligned}$$

Figure 4.6: Flops in the critical path during the matrix vector multiply

Updating the matrix vector product

Updating the matrix vector product, $y = y - VW^Tv - WV^Tv$, requires four matrix vector multiplies. $\mathsf{temp} = W^Tv$ and $\mathsf{temp} = V^Tv$ are both $(n-j)\times j'$ by $n-j$ matrix vector multiplies, where $j' = j \mod nb$. Both the matrix and the vector are stored in the current process column. No data movement is required to perform the computation, however the result, a vector of length $j'-1$ is the sum of the matrix vector multiplies performed on each of the processes in the process column.

Table 4.4: The cost of updating the matrix vector product in `PDLATRD`(Line 4.1 in Figure 4.5)

Task	File:line number or subroutine	Execution time contribution from columns $j = 1$ to n shown explicitly	Execution time (simplified)
Broadcast W^T unnecessarily for **temp** $= W^T v$	*pdlatrd.f*:373 *pbdgemv.f*:826 *dgebs2d*	$\sum_{j=1}^{n}\left(\delta_4+\lceil\log_2(p_c)\rceil\alpha+\lceil\log_2(p_c)\rceil\frac{(n-j)}{p_r}\beta+\lceil\log_2(p_c)\rceil\frac{\mathsf{nb}}{2}\beta\right)$	$n\,\delta_4+n\lceil\log_2(p_c)\rceil\alpha+0.5\frac{n^2\lceil\log_2(p_c)\rceil}{p_r}\beta+0.5n\,\mathsf{nb}\lceil\log_2(p_c)\rceil\beta$
Local computation of **temp** $= W^T v$	*pdlatrd.f*:373 *pbdgemv.f*:846 *dgemv*	$\sum_{j=1}^{n}\left(\delta_2+2\frac{n}{p_r}\frac{\mathsf{nb}}{2}\gamma_2\right)$	$n\,\delta_2+0.5\frac{n^2\,\mathsf{nb}}{p_r}\gamma_2$
Sum the contribution of **temp** from all processes in the column	*pdlatrd.f*:373 *pbdgemv.f*:858 *dgsum2d*	$\sum_{j=1}^{n}\left(\lceil\log_2(p_r)\rceil\alpha+\lceil\log_2(p_r)\rceil\frac{\mathsf{nb}}{2})\beta\right)$	$n\lceil\log_2(p_r)\rceil\alpha+0.5\,n\,\mathsf{nb}\lceil\log_2(p_r)\rceil\beta$
Broadcast **temp** (row-wise) to all processes in this column	*pdlatrd.f*:376 *pbdgemv.f*:579 *dgebs2d*	$\sum_{j=1}^{n}\left(\delta_4+\lceil\log_2(p_r)\rceil\alpha+\lceil\log_2(p_r)\rceil\frac{\mathsf{nb}}{2}\beta\right)$	$n\,\delta_4+n\lceil\log_2(p_r)\rceil\alpha+0.5\,n\,\mathsf{nb}\lceil\log_2(p_r)\rceil\beta$
Local computation of $y = V\cdot$ **temp**	*pdlatrd.f*:376 *pbdgemv.f*:600 *dgemv*	$\sum_{j=1}^{n}\left(\delta_2+2\frac{n}{p_r}\frac{\mathsf{nb}}{2}\gamma_2\right)$	$n\,\delta_2+0.5\frac{n^2\,\mathsf{nb}}{p_r}\gamma_2$
$y = y + WV^T$ is identical to $y = y + VW^T$	*pdlatrd.f*:379 *pdlatrd.f*:382	$n\lceil\log_2(p_c)\rceil\alpha+2\,n\lceil\log_2(p_r)\rceil\alpha+0.5\frac{n^2\lceil\log_2(p_c)\rceil}{p_r}\beta+0.5n\,\mathsf{nb}\lceil\log_2(p_c)\rceil\beta+n\,\mathsf{nb}\lceil\log_2(p_r)\rceil\beta+2n\,\delta_2+\frac{n^2\,\mathsf{nb}}{p_r}\gamma_2+2n\,\delta_4$	
Total	$2n\lceil\log_2(p_c)\rceil\alpha+4n\lceil\log_2(p_r)\rceil\alpha+\frac{n^2\lceil\log_2(p_c)\rceil}{p_r}\beta+n\,\mathsf{nb}\lceil\log_2(p_c)\rceil\beta+2n\,\mathsf{nb}\lceil\log_2(p_r)\rceil\beta+4n\,\delta_2+2\frac{n^2\,\mathsf{nb}}{p_r}\gamma_2+4n\,\delta_4$		
Standard data layout (See section 4.2.2)	$6\,n\lceil\log_2(\sqrt{p})\rceil\alpha+\frac{n^2\lceil\log_2(\sqrt{p})\rceil}{p_r}\beta+3n\,\mathsf{nb}\lceil\log_2(\sqrt{p})\rceil\beta+4n\,\delta_2+2\frac{n^2\,\mathsf{nb}}{\sqrt{p}}\gamma_2+4n\,\delta_4$		

The other two matrix vector multiplies, $y = V \cdot$ **temp** and $y = W \cdot$ **temp**, are both $(n-j) \times (j'-1)$ by $j'-1$ matrix vector multiplies. Again, the computation is performed entirely within the current process column. The 1 by $j'-1$ vector, **temp**, must be spread down, i.e. broadcast column-wise, to all processes in this process column, however no further communication is necessary in order to update y, as y is perfectly aligned with V.

Details are given in the table 4.4.

Computing the companion update vector

The details involved in computing the companion update vector are shown in table 4.5.

Table 4.5: The cost of computing the companion update vector in PDLATRD (Line 5.1 in Figure 4.5)

Task	File:line number or subroutine	Execution time contribution from columns $j = 1$ to n shown explicitly	Execution time (simplified)
Compute $y = \tau\, y$	*pdlatrd.f*:385 *pdscal*	$\sum_{i=1}^{n} \frac{1}{3}\,\delta_4$	$\frac{1}{3}\,n\,\delta_4$
Compute $\alpha = -0.5\tau\, y^T v$	*pdlatrd.f*:386 *pddot*	$\sum_{i=1}^{n} (\lceil \log_2(p_r) \rceil\, \alpha + \frac{1}{3}\delta_4)$	$n\,\lceil \log_2(p_r) \rceil\, \alpha + \frac{1}{3} n\,\delta_4$
Compute $w = y - \alpha v$	*pdlatrd.f*:390 *pdaxpy*	$\sum_{i=1}^{n} (\lceil \log_2(p_r) \rceil\, \alpha + \frac{1}{3}\delta_4)$	$n\,\lceil \log_2(p_r) \rceil\, \alpha + \frac{1}{3} n\,\delta_4$
Total	$2\,n\,\lceil \log_2(p_r) \rceil\, \alpha + n\,\delta_4$		
Standard data layout (See section 4.2.2)	$2n\,\lceil \log_2(\sqrt{p}) \rceil\, \alpha + n\,\delta_4$		

Performing the rank 2k update

The rank 2-k update is performed once per block column (i.e. n/nb times):

$$A = A - v w^T - w v^T \ .$$

PDSYTRD broadcasts v and w along processor tows, transposes them and then broadcasts them along processor columns. I ignore the α (latency) cost of the transpose here, because it is less significant (by a factor of nb) than the similar cost for the transpose in the matrix-vector multiply and because it is only relevant when $\frac{\mathrm{lcm}(p_r, p_c)}{p_c}$ is very large. The third β term in the transpose and broadcast operation should be multiplied by $\frac{\frac{\mathrm{lcm}(p_r,p_c)}{p_c} - 1}{\frac{\mathrm{lcm}(p_r,p_c)}{p_c}}$ but the added complexity is not justified for a small term.

The number of flops performed during the rank two update of $A(j : n, j : n)$ is modeled as:

$$2 \times 2 \times \mathsf{nb}\,(\frac{1}{2}(\frac{n-j}{p_r} + \frac{\mathsf{nb}}{2})(\frac{n-j}{p_c} + \frac{\mathsf{nb}}{2}) + \frac{n\,\mathsf{nb}\,\mathsf{pbf}}{2\,p_r}) \ .$$

The number of flops performed per matrix element involved in the rank-2 update is $2\times2\times\mathsf{nb}$. The number of elements in the lower triangular matrix is given by the sum of the terms within the parentheses.

The total number of flops for all rank two updates is modeled as the sum of this quantity as j ranges from nb to n by nb.

Table 4.6: The cost of performing the rank-$2k$ update (PDSYR2K) (Lines 6.1 through 6.3 in Figure 4.5)

Task	File:line number or subroutine	Execution time contribution from columns $j=1$ to n shown explicitly	Execution time (simplified)
Broadcast V and W within process rows (Line 6.1)	*pdsytrd.f*:354 *pdsyr2k_.c* *pdsyr2k.f*:454,477 *dgebs2d*	$\sum_{j=\mathsf{nb},\mathsf{nb}}^{n} \big(2\lceil\log_2(p_c)\rceil\alpha + 2(\frac{n-j}{p_r}+\frac{\mathsf{nb}}{2})\lceil\log_2(p_c)\rceil\beta\big)$	$2\frac{n}{\mathsf{nb}}\lceil\log_2(p_c)\rceil\alpha$ $+\frac{n^2}{p_r}\lceil\log_2(p_c)\rceil\beta$ $-n\,\mathsf{nb}\lceil\log_2(p_c)\rceil\beta$
Transpose and broadcast V and W within process columns (Line 6.2)	*pdsytrd.f*:354 *pdsyr2k_.c* *pdsyr2k.f*:491,847 *pbdtran*	$\sum_{j=\mathsf{nb},\mathsf{nb}}^{n} \big(2\lceil\log_2(p_r)\rceil\alpha + 2(\frac{n-j}{p_c}+\frac{\mathsf{nb}}{2})\mathsf{nb}\lceil\log_2(p_r)\rceil\beta + \frac{n-j}{p_c}\beta\big)$	$2\frac{n}{\mathsf{nb}}\lceil\log_2(p_r)\rceil\alpha$ $+\frac{n^2}{p_c}\lceil\log_2(p_r)\rceil\beta$ $-n\,\mathsf{nb}\lceil\log_2(p_r)\rceil\beta$ $+\frac{n^2}{p_c}\beta$
$\mathrm{tril}(A,0)=\mathrm{tril}(A,0)$ $+V\cdot W^T+W\cdot V^T$ (Line 6.3)	*pdsytrd.f*:354 *pdsyr2k_.c* *pdsyr2k.f*:655–60 , 1052–57 *pdgemm*	$\sum_{j=\mathsf{nb},\mathsf{nb}}^{n} \big(4\frac{n-j}{\mathsf{nb}\,p_c\,\mathsf{pbf}}\delta_3+4\mathsf{nb}\times(\frac{1}{2}(\frac{n-j}{p_r}+\frac{\mathsf{nb}}{2})(\frac{n-j}{p_c}+\frac{\mathsf{nb}}{2})+\frac{n\,\mathsf{nb}\,\mathsf{pbf}}{2\,p_r})\gamma_3\big)$	$2\frac{n^2}{\mathsf{nb}^2\,p_c\,\mathsf{pbf}}\delta_3+$ $\frac{2}{3}\gamma_3+\frac{1}{2}\frac{n^2\,\mathsf{nb}}{p_r}\gamma_3+$ $\frac{1}{2}\frac{n^2\,\mathsf{nb}}{p_c}\gamma_3+\frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{p_r}\gamma_3$
Total	$2\frac{n}{\mathsf{nb}}\lceil\log_2(p_c)\rceil\alpha+2\frac{n}{\mathsf{nb}}\lceil\log_2(p_r)\rceil\alpha+\frac{n^2}{p_r}\lceil\log_2(p_c)\rceil\beta+\frac{n^2}{p_c}\lceil\log_2(p_r)\rceil\beta+\frac{n^2}{p_c}\beta$ $-n\,\mathsf{nb}\lceil\log_2(p_c)\rceil\beta-n\,\mathsf{nb}\lceil\log_2(p_r)\rceil\beta+2\frac{n^2}{\mathsf{nb}^2\,p_c\,\mathsf{pbf}}\delta_3+\frac{2}{3}\gamma_3+\frac{1}{2}\frac{n^2\,\mathsf{nb}}{p_r}\gamma_3$ $+\frac{1}{2}\frac{n^2\,\mathsf{nb}}{p_c}\gamma_3+\frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{p_r}\gamma_3$		
Standard data layout (See section 4.2.2)	$4\frac{n}{\mathsf{nb}}\lceil\log_2(\sqrt{p})\rceil\alpha+2\frac{n^2}{\sqrt{p}}\lceil\log_2(\sqrt{p})\rceil\beta+\frac{n^2}{\sqrt{p}}\beta$ $-2n\,\mathsf{nb}\lceil\log_2(\sqrt{p})\rceil\beta+\frac{n^2}{\mathsf{nb}^2\sqrt{p}}\delta_3+\frac{2}{3}\gamma_3+3\frac{n^2\,\mathsf{nb}}{\sqrt{p}}\gamma_3$		

The negative term $(-2\frac{n^2\,\mathsf{nb}}{p}\gamma_3)$, which results from the fact that j starts at nb, is ignored because it is $O(\frac{n^2\,\mathsf{nb}}{p})$ and hence too small.

Details are given in table 4.6.

4.2.3 PDSYTRD execution time summary

Table 4.7 shows that the computation cost in PDSYTRD is:

$$\frac{2}{3}\frac{n^3}{p}\gamma_3 + \frac{2}{3}\frac{n^3}{p}\gamma_2 + \\ \frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{p_r}\gamma_2 + \frac{7}{2}\frac{n^2\,\mathsf{nb}}{p_r}\gamma_2 + \frac{1}{2}\frac{n^2\,\mathsf{nb}}{p_c}\gamma_2 + \frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{p_r}\gamma_2 + \\ \frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{p_r}\gamma_3 + \frac{1}{2}\frac{n^2\,\mathsf{nb}}{p_r}\gamma_3 + \frac{1}{2}\frac{n^2\,\mathsf{nb}}{p_c}\gamma_3 + \\ 2\frac{n^2}{\mathsf{nb}^2\,p_c\,\mathsf{pbf}}\delta_3 + 6\,n\,\delta_2 + 2\frac{n^2}{p_r\,\mathsf{pbf}\,\mathsf{nb}}\delta_2 + \frac{n^2}{p_c}\delta_1 + 9\,n\,\delta_4\ .$$

The most important terms in the computation cost are the $O(\frac{n^3}{p})$ flops. The relative importance of the other ($o(n^3)$) terms depends on the computer. On the PARAGON none stand out above the rest. Indeed on the PARAGON none of the $o(n^3)$ terms accounts for more than 3% of the total execution time of PDSYEVX when $n = 3480$ and $p = 64$. However, all of $o(n^3)$ terms combined account for 21% of the total execution time on that same problem.

Figure 4.8 shows that the computation cost in the tridiagonal eigendecomposition in PDSYEVXis:

$$53\frac{ne}{p}\gamma_{\div} + 3\frac{nm}{p}\gamma_{\div} + 112n\gamma_{\div} + 265\frac{ne}{p}\gamma_1 + 45\frac{nm}{p}\gamma_1 + 620n\gamma_1 + 6nc^2\gamma_1\ .$$

The execution time of tridiagonal eigendecomposition is dominated by the cost of divides, and the size of the largest cluster, c. The load imbalance terms ($112n\gamma_{\div}$ and $620n\gamma_1$) are neglible.

Table 4.9 shows that the communication cost in PDSYTRD is:

$$4n\,\lceil\log_2(p_c)\rceil\alpha + 13n\,\lceil\log_2(p_r)\rceil\alpha + n\ \mathrm{lcm}(p_r, p_c)/p_r\alpha + \\ n\,\lceil\log_2(\mathrm{lcm}(p_r, p_c))\rceil\alpha + \\ 3\frac{n^2}{p_r}\lceil\log_2(p_c)\rceil\beta + 2\frac{n^2}{p_c}\lceil\log_2(p_r)\rceil\beta + \frac{1}{2}\frac{n^2}{p_r}\beta + 2\frac{n^2}{p_c}\beta\ .$$

Most of the messages are in broadcasts and reductions (i.e. the $O(n\log(p))$ terms) and most of the broadcasts and reductions ($13n$) are within processor rows, versus only $4n$ broadcasts and reductions within processor columns. By contrast, the message volume is fairly evenly split between broadcasts and reductions within processor rows (

$3\,\frac{n^2}{p_r}\,\lceil \log_2(p_c) \rceil \beta$) and broadcasts and reductions within processor columns ($2\,\frac{n^2}{p_c}\,\lceil \log_2(p_r) \rceil \beta$).

The lcm terms are negligible unless p is very large, in which case it is important to make sure that $\mathrm{lcm}(p_r, p_c)$ is reasonable ($say < 10 \max(p_r, p_c)$).

4.3 Eigendecomposition of the tridiagonal

The execution time of tridiagonal eigendecomposition is dominated by two factors: the size of the largest cluster of eigenvalues and the speed of the divide.

4.3.1 Bisection

During bisection, in DSTEBZ, each Sturm count requires n divisions and $5n$ other flops to produce one additional bit of accuracy. Hence, it takes roughly $53\,n$ divisions and $53 \times 5\,n$ flops[9] for each eigenvalue and $53\,n\,e$ total divisions for all eigenvalues in IEEE double precision, where e is the number of eigenvalues to be computed. The exact number of divisions and flops depends on the actual eigenvalues, the parallelization strategy and other factors. However, this simple model suffices for our purposes.

4.3.2 Inverse iteration

Inverse iteration typically requires $3n$ divides and $45n$ flops per eigenvalue plus the cost of re-orthogonalization.

In PDSYEVX the number of flops performed by any particular processor, p_i, during re-orthogonalization is: $\sum_{C \in \{\text{clusters assigned to } p_i\}} 4 \sum_{i=1}^{\mathrm{size}(C)} \mathsf{n\_iter}(i)\, n\,(i-1)$. Where: $\mathsf{n\_iter}(i)$ is the number of inverse iterations performed for eigenvalue i (typically 3). If the size of the largest cluster is greater than $\frac{n}{p}$, the processor which is responsible for this cluster will not be responsible for any eigenvalues outside of this cluster.

Hence, if the size of the largest cluster is greater than $\frac{n}{p}$, the number of flops performed by the processor to which this processor is assigned is (on average):

$$4\,\mathsf{n\_iter}\,n\,\frac{1}{2}c^2 = 6n\,c^2$$

[9] Although these are not all BLAS Level 1 flops, they have the same ratio memory operations to flops that are typical of BLAS Level 1 operations.

where: $c = \max_{C \in \{\text{clusters}\}} \text{size}(C)$ i.e. the number of eigenvalues in the largest cluster, and $\mathsf{n\_iter} = 3$ is the average number of inverse iterations performed for each eigenvalue.

As the problem size and number of processors grows, the largest cluster that `PDSYEVX` is able to reorthogonalize properly gets smaller (relative to n). As a consequence, reorthogonalization will not require large execution time[10] Specifically, if the largest cluster has fewer than $\frac{n}{p}$ eigenvalues, (i.e. fits easily on one processor) the number of eigenvalues that will be assigned to any one processor, and hence the total number of flops it must perform, is limited. The worst case is where there are $p+1$ clusters each of size $\frac{n}{p+1}$. In this case, one processor must be assigned 2 clusters of size $\frac{n}{p+1}$, requiring (on average) $2 \times 6\, n\, (\frac{n}{p+1})^2$ or roughly $12 \frac{n^3}{p^2}$. [11]

Our model for the execution time of Gram Schmidt re-orthogonalization ($\sum_{i=1}^{c} 4\, n\, i = 2\, n\, c^2 \gamma_1$, where c is the size of the largest cluster.) assumes that the processor to which the largest cluster is assigned is not assigned any other clusters. This is true if the largest cluster has more than n/p eigenvalues in it. If the largest cluster of eigenvalues contains fewer than n/p eigenvalues, reorthogonalization is relatively unimportant.

Inderjit Dhillon, Beresford Parlett and Vince Fernando's recent work[139, 77] on the tridiagonal eigenproblem substantially reduces the motivation to model the existing `ScaLAPACK` tridiagonal eigensolution code in great detail, since we expect them to replace the current code with something that costs $O(\frac{n^2}{p})$ flops, $O(\frac{n^2}{p})$ message volume and $O(p)$ messages, which is negligible compared to tridiagonal reduction.

4.3.3 Load imbalance in bisection and inverse iteration

Load imbalance during the tridiagonal eigendecomposition is caused in part by the fact that not all processes will be assigned the same number of eigenvalues and eigenvectors and in part by the fact that different eigenvalues and eigenvectors will require slightly different amounts of computation. Our experience indicates that the load imbalance corresponds roughly to the cost of finding two eigenvalues $(2 \times (53 n \gamma_{\div} + 53 \times 5 n \gamma_1))$ and two eigenvectors $(2 \times (3 n \gamma_{\div} + 45 n \gamma_1))$ on one processor. Hence, our execution time model for the load imbalance during tridiagonal eigedecomposition is: $((2 \times 53 + 2 \times 3) =$

[10] This is not to suggest that reorthogonalization in `PDSYEVX` gets better as n and p increase. (indeed `PDSYEVX` may fail to reorthogonalize large clusters for large n and p) It just means that reorthogonalization in `PDSYEVX` will not take a long time for large n and large p.

[11] The appearance of p^2 in the denominator stems from the restriction $c \leq \frac{n}{p}$, meaning that as p increases the largest cluster size that `PDSYEVX` can handle efficiently decreases.

$112n\gamma_{\div} + (2 \times 53 \times 5 + 2 \times 45) = 620n\gamma_1)$

In evaluating the cost of load imbalance in tridiagonal eigendecomposition, one must include load imbalance in Gram Schmidt reorthogonalization. Indeed if the input matrix has one cluster of eigenvalues that is substantially larger than all others (yet small enough to fit on one processor so that `PDSYEVX` can reorthogonalize it) Gram Schmidt reorthogonalization is very poorly load balanced and could be treated almost enttirely as a load imbalance cost.

We do not separate the load imalance cost of Gram Schmidt from what the execution time for Gram Schmidt would be if the load were balanced, because doing so would complicate the model without making it match actual execution time any better.

4.3.4 Execution time model for tridiagonal eigendecomposition in `PDSYEVX`

The cost of tridiagonal eigedecomposition in `PDSYEVX` is the sum of the cost of bisection, inverse iteration and reorthogonalization. Hence:

$$53\,\frac{n\,e}{p}\,\gamma_{\div} + 3\,\frac{n\,m}{p}\,\gamma_{\div} + 112\,n\,\gamma_{\div} + 265\,\frac{n\,e}{p}\,\gamma_1 + 45\,\frac{n\,m}{p}\,\gamma_1 + 620\,n\,\gamma_1 + 2\,n\,c^2\,\gamma_1$$

The load imbalance terms $112\,n\,\gamma_{\div}$ and $530\,n\,\gamma_1$ stem partly from the fact that some processors will typically be assigned at least one more eigenvalue and/or eigenvector than other processors and from the fact that both bisection and inverse iteration are iterative procedures requiring more time on some eigenvalues than on others.

4.3.5 Redistribution

Inverse iteration typically leaves the data distributed in a manner in which it would be awkward and inefficient to perform back transformation. If each eigenvector is computed entirely within one processor, as `PDSTEIN` does, inverse iteration requires no communication, provided that all processors have a copy of the tridiagonal matrix and the eigenvalues. This, however, leaves the eigenvector matrix distributed in a one-dimensional manner in which back transformation would be inefficient. Furthermore, since different processors may have been assigned to compute a different number of eigenvectors (to improve orthogonality among the eigenvectors) the eigenvector matrix will typically not be distributed in a block cylic manner. Since `PDORMTR` (and all `ScaLAPACK` matrix transformations) requires that the data be in a 2D block cyclic distribution, the eigenvectors must,at least, be redistributed to a block cyclic distribution. For convenience and potential efficiency[12], PDSTEIN redistributes the eigenvector matrix.

The simplest method of data redistribution is to have each processor send one message to each of the other processors. That message contains the data owned by the sender and needed by the receiver. Redistributing the data in this manner requires that each processor send every element that it owns to other processors[13] and receive what it needs from other processors. Since each processor owns[14] roughly $(n\,m)/p$ elements and needs roughly $(n\,m)/p$ elements, the total data sent and received by each processor is roughly $2(n\,m)/p$. In our experience, data redistribution is slightly less efficient than other broadcasts and reductions and hence we use $4(n\,m)/p\beta$ as our model for the data redistribution cost.

4.4 Back Transformation

Transforming the eigenvectors of the tridiagonal matrix back to the eigenvectors of the original matrix requires multiplying a series of Householder vectors. The Householder updates can be applied in a blocked manner with each update taking the form: $(I+VTV^T)$, where $V \in R^{n',\mathsf{nb}}$ is the matrix of Householder vectors, and T is an $(\mathsf{nb} \times \mathsf{nb}$ a triangular matrix[27].

The following steps compute $Z' = (I + VTV^T)Z$. These are performed for each block Householder update. The major contributors to the cost are noted below.

Compute T

Computing the nb by nb triangular matrix T requires nb calls to DGEMV, a summation of $\mathsf{nb}^2/2$ elements within the current processor column and nb calls to DTRMV. The computation of T need not be in the critical path. There are n/nb different matrices T that need to be computed, and they could be computed in advance in parallel.

Compute $W = V^T Z$.

Spread V across. Compute $V^T Z$ locally. Sum W within each processor column.

[12] The actual efficiency depends upon the data distribution chosen by the user for the input and output matrices

[13] Although some data will not have to be sent because it is owned and needed by the same processor, this will typically be a minor savings.

[14] In the absence of large clusters of eigenvalues assigned to a single processor.

The spread across of V is performed on a ring topology because the processor columns need not be synchronized. Each processor column must receive V and send V, hence the cost for each processor column is: $(2\, n' \, \mathsf{nb})/p_r$

The local computation of $V^T Z$ is a call to `DGEMM` involving $2(m/p_c + \mathsf{vnb})(n'/p_r + \mathsf{nb}/2)\mathsf{nb}$ flops. Ignoring the lower order $\mathsf{vnb}\,\mathsf{nb}^2$ term, this is:

$$2\,(n'\, m\, \mathsf{nb})/p + 2\,(n'\, \mathsf{vnb})/p_r + 2\,(m\, \mathsf{nb})/p_c\ .$$

Compute $W = TW$

Local.

Compute $Z = Z - VW$

Spread W down. Local computation. (Note: V has already been spread across.)

The local computation of $Z - VW$, like the computation of VTZ involves a call to `DGEMM` involving $2(m/p_c + \mathsf{vnb})(n'/p_r + \mathsf{nb}/2)\mathsf{nb}$ flops.

Back transformation differs from reduction to tridiagonal form in many ways. It requires many fewer messages: $O(n/\mathsf{nb})$ versus $O(n)$. Because the back transformation of each eigenvector is independent, the Householder updates can be applied in a pipelined manner, allowing V to be broadcast in a ring instead of a tree topology. `PDLARFB` does not use the `PBLAS`, allowing V to be broadcast once but used twice. Since the number of eigenvectors does not change during the update, half of the load imbalance depends on $\bmod\,(n, \mathsf{nb}\, p_c)$ and can be reduced significantly if $\bmod\,(n, \mathsf{nb}\, p_c) = 0$. In the following table vnb is the imbalance in the 2D block-cyclic distribution of eigenvectors[15]

The cost of back transformation, shown in table 4.10, is asymmetric, the $(O(n^2/p_r))$ cost is smaller than the $(O(n^2/p_c))$ cost. Furthermore, the $(O(n^2/p_r))$ cost can be reduced further by computing T in parallel, and choosing a data layout which will minimze vnb. Reducing the $O(n^2/p_r)$ cost would allow $p_r < p_c$, reducing the $O(n^2/p_c)$ costs. This is discussed further in Chapter 8.

[15] vnb is computed as follows: $extravecsonproc1 - extravecs/p_r$. Where: $extravecs = \bmod\,(n, \mathsf{nb}\, p_c)$ and $extravecsonproc1 = \min(\mathsf{nb}, extravecs)$.

Table 4.7: Computation cost in PDSYEVX

scale factor	update current column (Table 4.1)	compute reflector (Table 4.2)	matrix vector product (Table 4.4)	update vector product (Table 4.5)	compute update vector (Table 4.6)	perform rank $2k$ update (Table 4.10)	tridiagonal eigendecomposition (Section 4.3)	back transformation (Table 4.10)	total
$\frac{n^3}{p}\ \gamma_3$						$\frac{2}{3}$			$\frac{2}{3}$
$\frac{n^2 m}{p}\ \gamma_3$								2	2
$\frac{n^3}{p}\ \gamma_2$			$\frac{2}{3}$						$\frac{2}{3}$
$\frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{p_r}\ \gamma_2$			1						1
$\frac{n^2\,\mathsf{nb}}{p_r}\ \gamma_2$	1		$\frac{1}{2}$	2				$\frac{1}{2}$	4
$\frac{n^2\,\mathsf{nb}}{p_c}\ \gamma_2$			$\frac{1}{2}$						$\frac{1}{2}$
$\frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{p_r}\ \gamma_3$						1			1
$\frac{n^2\,\mathsf{nb}}{p_r}\ \gamma_3$						$\frac{1}{2}$			$\frac{1}{2}$
$\frac{n^2\,\mathsf{vnb}}{p_r}\ \gamma_3$								2	2
$\frac{n^2\,\mathsf{nb}}{p_c}\ \gamma_3$						$\frac{1}{2}$			$\frac{1}{2}$
$\frac{n\,m\,\mathsf{nb}}{p_c}\ \gamma_3$								3	3
$\frac{n}{\mathsf{nb}}\ \delta_3$								3	3
$\frac{n^2}{\mathsf{nb}^2\,p_c\,\mathsf{pbf}}\ \delta_3$						2			2
$n\ \delta_2$	2			4				2	8
$\frac{n^2}{p_r\,\mathsf{pbf}\,\mathsf{nb}}\ \delta_2$			2						2
$\frac{n^2}{p_c}\ \delta_1$				1					1
$n\delta_4$	2	1	1	4	1				9

Table 4.8: Computation cost (tridiagonal eigendecomposition) in `PDSYEVX`

scale factor	update current column (Table 4.1)	compute reflector (Table 4.2)	matrix vector product (Table 4.4)	update vector product (Table 4.5)	compute update vector (Table 4.6)	perform rank $2k$ update (Table 4.10)	tridiagonal eigendecomposition (Section 4.3)	back transformation (Table 4.10)	total
$\frac{ne}{p}\gamma_\div$							53		53
$\frac{nm}{p}\gamma_\div$							3		3
$n\gamma_\div$							112		112
$\frac{ne}{p}\gamma_1$							265		265
$\frac{nm}{p}\gamma_1$							45		45
$n\gamma_1$							620		620
$nc^2\gamma_1$							6		6

Table 4.9: Communication cost in PDSYEVX

scale factor	update current column (Table 4.1)	compute reflector (Table 4.2)	matrix vector product (Table 4.4)	update vector product (Table 4.5)	compute update vector (Table 4.6)	perform rank $2k$ update (Table 4.10)	tridiagonal eigendecomposition (Section 4.3)	back transformation (Table 4.10)	total
$n \lceil \log_2(p_c) \rceil \alpha$			2	2					4
$n \lceil \log_2(p_r) \rceil \alpha$	2	3	2	4	2				13
$n \ \mathrm{lcm}(p_r, p_c)/p_r \alpha$			1						1
$n \ \mathrm{lcm}(p_r, p_c)/p_c \alpha$			1						1
$n \lceil \log_2(\mathrm{lcm}(p_r, p_c)) \rceil \alpha$			1						1
$\frac{n^2}{p_r} \lceil \log_2(p_c) \rceil \beta$			1	1		1			3
$\frac{n^2}{p_c} \lceil \log_2(p_r) \rceil \beta$			1			1			2
$\frac{n^2}{p_r} \beta$			$\frac{1}{2}$						$\frac{1}{2}$
$\frac{n^2}{p_c} \beta$			1			1			2
$n \, \mathsf{nb} \lceil \log_2(p_c) \rceil \beta$			1	1		-1			1
$n \, \mathsf{nb} \lceil \log_2(p_r) \rceil \beta$	1		1	2		-1			3

Table 4.10: The cost of back transformation (`PDORMTR`)

Task	File:line number or subroutine	Execution time contribution from columns $j = 1$ to n shown explicitly	Execution time (simplified)
Compute T	*pdsyevx.f*:855 *pdormtr.f*:408 *pdormqr.f*:394 *pdlarft.f*:	$\sum_{n'=1,\mathsf{nb}}^{n} \big(\lceil \log_2(p_r) \rceil \frac{\mathsf{nb}^2}{2} \beta +$ $2\,\mathsf{nb}\,\delta_2 + 2 \frac{n'\,\mathsf{nb}^2}{2 p_r} \gamma_2 + \big)$	$2\,n\,\delta_2 + 0.5 \frac{n^2\,\mathsf{nb}}{p_r} \gamma_2$
Compute $W = V^T Z$	*pdsyevx.f*:855 *pdormtr.f*:408 *pdormqr.f*:412 *pdlarfb.f*:322, 398,405	$\sum_{n'=1,\mathsf{nb}}^{n} \big(2 \frac{n'\,\mathsf{nb}}{p_r} \beta$ $+ \lceil \log_2(p_r) \rceil \frac{m\,\mathsf{nb}}{p_c} \beta + \delta_3$ $+ 2 \frac{m\,\mathsf{nb}^2}{2\,p_c} \gamma_3 + 2 \frac{\mathsf{vnb}\,n'\,\mathsf{nb}}{p_r} \gamma_3$ $+ 2 \frac{n\,n'\,\mathsf{nb}}{p} \gamma_3 \big)$	$\frac{n^2}{p_r} \beta +$ $\frac{n\,m}{p_c} \lceil \log_2(p_r) \rceil \beta +$ $\frac{n}{\mathsf{nb}} \delta_3 + \frac{n\,m\,\mathsf{nb}}{p_c} \gamma_3 +$ $\frac{n^2\,\mathsf{vnb}}{p_r} \gamma_3 + \frac{n^2\,m}{p} \gamma_3$
Compute $W = TW$	*pdsyevx.f*:855 *pdormtr.f*:408 *pdormqr.f*:412 *pdlarfb.f*:412	$\sum_{n'=1,\mathsf{nb}}^{n} \big(\delta_3 + 2 \frac{m\,\mathsf{nb}^2}{2\,p_c} \gamma_3 \big)$	$\frac{n}{\mathsf{nb}} \delta_3 + \frac{n\,m\,\mathsf{nb}}{p_c} \gamma_3$
Compute $Z = Z - VW$	*pdsyevx.f*:855 *pdormtr.f*:408 *pdormqr.f*:412 *pdlarfb.f*:415,425	$\sum_{n'=1,\mathsf{nb}}^{n} \big(\lceil \log_2(p_r) \rceil \frac{m\,\mathsf{nb}}{p_c} \beta$ $+ \delta_3 + 2 \frac{m\,\mathsf{nb}^2}{2\,p_c} \gamma_3 +$ $2 \frac{\mathsf{vnb}\,n'\,\mathsf{nb}}{p_r} \gamma_3 + 2 \frac{m\,n'\,\mathsf{nb}}{p} \gamma_3 \big)$	$\frac{n\,m}{p_c} \lceil \log_2(p_r) \rceil \beta +$ $\frac{n}{\mathsf{nb}} \delta_3 + \frac{n\,m\,\mathsf{nb}}{p_c} \gamma_3 +$ $\frac{n^2\,\mathsf{vnb}}{p_r} \gamma_3 + \frac{n^2\,m}{p} \gamma_3$
Total	$\frac{n^2}{p_r} \beta + 2 \frac{n\,m}{p_c} \lceil \log_2(p_r) \rceil \beta + 2\,n\,\delta_2 + 0.5 \frac{n^2\,\mathsf{nb}}{p_r} \gamma_2 + 3 \frac{n}{\mathsf{nb}} \delta_3 + 3 \frac{n\,m\,\mathsf{nb}}{p_c} \gamma_3$ $+ 2 \frac{n^2\,\mathsf{vnb}}{p_r} \gamma_3 + 2 \frac{n^2\,m}{p} \gamma_3$		
Standard data layout	$\frac{n^2}{\sqrt{p}} \beta + 2 \frac{n\,m}{\sqrt{p}} \lceil \log_2(\sqrt{p}) \rceil \beta + 2\,n\,\delta_2 + 0.5 \frac{n^2\,\mathsf{nb}}{\sqrt{p}} \gamma_2 + 3 \frac{n}{\mathsf{nb}} \delta_3 + 3 \frac{n\,m\,\mathsf{nb}}{\sqrt{p}} \gamma_3$ $+ 2 \frac{n^2\,\mathsf{vnb}}{\sqrt{p}} \gamma_3 + 2 \frac{n^2\,m}{p} \gamma_3$		

Chapter 5

Execution time of the ScaLAPACK symmetric eigensolver, PDSYEVX on efficient data layouts on the Paragon

The detailed execution time model gives us confidence that we understand the execution time of PDSYEVXİt explains performance on a wide range of problem sizes, data layouts, input matrices, computers and user requests. However, the same complexity that allows the detailed model to explain performance over such a large domain makes it difficult to grasp, understand and interpret. The simple six term model shown in this chapter is designed to explain the performance of the common, efficient case on a well known computer.

PDSYEVX takes 205 seconds to compute the eigendecomposition of a 3840 by 3840 symmetric random matrix on a 64 node Paragon in double precision. Counting only the $\frac{10}{3}n^3$ flops, PDSYEVX achieves 920 Gigaflops per second which equals 14 Megaflops per second per node.

For large, well behaved[1], matrices, PDSYEVX is efficient, as detailed in Table 5.1. For well behaved 3840×3840 matrices, PDSYEVX spends 63% = (28+35)% of its time on necessary computation and only 35% of its time on communication, load imbalance and

[1]For PDSYEVX's purpose, a well behaved matrix is one which does not have any large clusters of eigenvalues whose associated eigeventers must be computed orthogonally.

Table 5.1: Six term model for PDSYEVX on the Paragon

Component	Model	$n = 3840$, $p = 64$ % time
matrix transformation computation (See section 5.3)	$\frac{10}{3}\frac{n^3}{p}$ $(\gamma = .0215)$	35
tridiagonal eigendecomposition computation (See section 5.4)	$239\,\frac{n^2}{p}$	28
message initiation (See section 5.5)	$17\,n\,\log_2(\sqrt{p})$ $(\alpha = 65.9)$	10
message transmission (See section 5.6)	$7\frac{n^2}{\sqrt{p}}\log_2(\sqrt{p})$ $(\beta = .146)$	4
order n overhead & imbalance (See section 5.7)	$2780\,n$	7
order n^2 overhead & imbalance (See section 5.8)	$14.0\,\frac{n^2}{\sqrt{p}}$	14

overhead required for execution in parallel.

n Matrix size

p Number of processors

γ Matrix-matrix multiply time (= .0215 microseconds/flop)

α Message latency time (= 65.9 microseconds/message)

β Message throughput time (= .146 microseconds/word)

Although PDSYEVX is efficient on the PARAGON[2], Table 5.1 shows us that there is room for improvement. Ignoring the execution time required for solution of the tridiagonal eigenproblem for the moment, we note that the matrix transformations reach only about 50% of peak performance (35% vs. 35+10+4+7+14=72%) for this problem size (roughly the largest that will fit on this PARAGON). Furthermore, efficiency will be lower for smaller problem sizes.

Unfortunately, there is no single culprit that accounts for the inefficiency. Communication accounts for a bit less than half of the inefficiency, while software overhead accounts for a bit more than half of the inefficiency.

[2]Details about the hardware and software used for this timing run are given in table 6.3

One could argue that while n=3840 on 64 nodes is the largest problem that PDSYEVX can run on this particular computer, it is still a relatively small problem. However, there are several reasons not to ignore this result. First, while it is true that newer machines have more memory, they also have much faster floating point units, steeper memory hierarchies and few offer communication to computation ratios as high as the PARAGON. Furthermore, we should strive to achieve high efficiency across a range of problem sizes, not just for the largest problems that can fit on the computer. Achieving high efficiency on small problem sizes means that users can efficiently use more processors and hence reduce execution time.

In summary, PDSYEVX is a good starting point, but leaves room for improvement. However, significantly improving performance will require attacking more than one source of inefficiency.

The fact that PDSYEVX spends 28% of its total time in solving the tridiagonal eigenproblem is a result of the slow divide on the PARAGON. The PARAGON offers two divides: a fast divide and a slow divide that meets the IEEE 754 spec[7]. Although the ScaLAPACK's bisection and inverse iteration codes are designed to work with an inaccurate divide, ScaLAPACK uses the slow correct divide by default.

5.1 Deriving the PDSYEVX execution time on the Intel Paragon (common case)

This six term model is based on the detailed model described in section 4 which has been validated on a number of distributed memory computers and a wide range of data layouts and problem sizes.

5.2 Simplifying assumptions allow the full model to be expressed as a six term model

I assume that a reasonably efficient data layout is chosen. I set the data layout parameters as follows:

$\mathsf{nb} = 32$. The optimal block size on the Paragon is about 10, however the reduction in execution time obtained by using $\mathsf{nb} = 10$ rather than $\mathsf{nb} = 32$ is less than 10%, so we stick to our standard suggested value of nb.

$p_r = p_c = \sqrt{p}$. PDSYEVX achieves the best performance[3] when $p_c \leq p_r \leq \frac{p_c}{4}$. Assuming that $p_r = p_c = \sqrt{p}$ allows the p_r and p_c terms to be coalesced into a single $\sqrt{p}$ term.

pbf $= 2$. The panel blocking factor[4] , pbf $= \max(2, \mathrm{lcm}(p_r, p_c)/p_c)$ in ScaLAPACK version 1.5.

vnb $= 0$. vnb is the imbalance in the number of rows in the original matrix as distributed amongst the processors. I assume that the matrix is initially balanced perfectly amongst all processors, i.e. n is a multiple of p_r nb.

$\gamma_2 = \gamma_3$ We assume for the simplified model that all flops are performed at the peak flop rate. This introduces an error equal to $2/3\, n^3/p(\gamma_2 - \gamma_3)$ which is typically no more than 2-5% of the total time on the PARAGON.

$m = e = n$ Assume that a full eigendecomposition is required. i.e. all eigenvalues are required $e = n$ and all eigevectors are required $m = n$.

$c = 1$ Assume that the input matrix has no clusters of eigenvalues.

In addition, we set all of the machine parameters to constants measured or estimated on the Intel Paragon as shown in table 6.3 in order to coalesce the overhead, load imbalance, and tridiagonal eigen decomposition terms into just three terms.

5.3 Deriving the computation time during matrix transformations in PDSYEVX on the Intel Paragon

Table 5.2 shows that PDSYTRD performs $\frac{4}{3}\frac{n^3}{p} + O(n^2)$ flops per process. Of these, $\frac{2}{3}\frac{n^3}{p} + O(n^2)$ are matrix vector multiply flops and $\frac{2}{3}\frac{n^3}{p} + O(n^2)$ are matrix matrix multiply flops. PDSYTRD performs the same floating point operations that the LAPACK routine, DSYTRD, does. And $\frac{4}{3}n^3$ is the textbook[84] number of flops for reduction to tridiagonal form.

[3] Performance of PDSYEVX is not overly sensitive to the data layout, provided that nb is sufficiently large to allow good DGEMM performance, that the processor grid is reasonably close to square and that $\mathrm{lcm}(p_r, p_c)$ is not outrageous compared to p_c and p_r. (The latter factor is only relevant when one is dealing with thousands of processors.) I have not performed a detailed study of when using fewer processors results in lower execution time. However, if you drop processors only when necessary to make $p_c \leq p_r \leq \frac{p_c}{16}$ and $\mathrm{lcm}\, p_r, p_c \leq 10 p_c$ the processor grid chosen will allow performance within 10% of the optimal processor grid.

[4] The matrix vector multiplies are each performed in panels of size pbfnb. See Section 4.2.2.

Table 5.2: Computation time in PDSYEVX

Task	Full model	Six term model
computation time during reduction to tridiagonal form (See section 4.2)	$\frac{2}{3}\frac{n^3}{p}\gamma_2+\frac{2}{3}\frac{n^3}{p}\gamma_3$	$\frac{4}{3}\frac{n^3}{p}\gamma_3$
computation time during back transformation (See table 4.10)	$2\frac{n^2\,m}{p}\gamma_3$	$2\frac{n^3}{p}\gamma_3$
Total		$\frac{10}{3}\frac{n^3}{p}\gamma_3$

Table 5.3: Execution time during tridiagonal eigendecomposition

Task	Full model	Paragon model	Paragon time
computation time during tridiagonal eigendecomposition (See section 4.3)	$265\frac{n\,e}{p}\gamma_1+45\frac{n\,m}{p}\gamma_1+$ $53\frac{n\,e}{p}\gamma_\div+3\frac{n\,m}{p}\gamma_\div+$ $2n\,c^2\,\gamma_1$	$(310\times.074$ $+56\times3.85+0)\frac{n^2}{p}$	$239.\,\frac{n^2}{p}$
Total			$239.\,\frac{n^2}{p}$

PDORMTR performs $2\frac{n^3}{p}+O(n^2)$ flops per process. Again this is the same as the LAPACK routine.

5.4 Deriving the computation time during eigendecomposition of the tridiagonal matrix in PDSYEVX on the Intel Paragon

The computation time during tridiagonal eigendecomposition, in the absence of clusters of eigenvalues is $O(n^2)$ and hence for large n becomes less important.

The simplified model for the execution time of the tridiagonal eigensolution on the PARAGON in table 5.3 is obtained from the detailed model by replacing γ_1 and $\gamma_\div$ with their values on the PARAGON and by assuming that all clusters of eigenvalues are of modest size.

Load imbalance during the tridiagonal eigendecomposition is caused in part by the fact that not all processes will be assigned the same number of eigenvalues and eigenvectors and in part by the fact that different eigenvalues and eigenvectors will require slightly different amounts of computation. Our experience indicates that the load imbalance corresponds roughly to the cost of finding two eigenvalues and two eigenvectors.

Table 5.4: Message initiations in PDSYEVX

Task	Full model	Six term model
message initiation during reduction to tridiagonal form (See table 4.9)	$(13\lceil\log_2(p_r)\rceil+4\lceil\log_2(p_c)\rceil)\, n\, \alpha$	$17\, n\, \log_2(\sqrt{p})\, \alpha$
Total		$17\, n\, \log_2(\sqrt{p})\, \alpha$

Table 5.5: Message transmission in PDSYEVX

Task	Full model	Six term model
message transmission time during reduction to tridiagonal form (See table 4.9)	$(3\lceil\log_2(p_c)\rceil \frac{n^2}{p_r}+2\lceil\log_2(p_r)\rceil \frac{n^2}{p_c})\beta$	$5\, \frac{n^2}{\sqrt{p}} \log_2(\sqrt{p})\, \beta$
message transmission time during back transformation (See table 4.10)	$2\lceil\log_2(p_r)\rceil \frac{n\, m}{p_c}\, \beta$	$2\, \frac{n^2}{\sqrt{p}} \log_2(\sqrt{p})\, \beta$
Total		$7\, \frac{n^2}{\sqrt{p}} \log_2(\sqrt{p})\, \beta$

5.5 Deriving the message initiation time in PDSYEVX on the Intel Paragon

Table 5.4 shows that PDSYEVX requires $17\, n\, \log(\sqrt{p})$ message initiations.

5.6 Deriving the inverse bandwidth time in PDSYEVX on the Intel Paragon

Table 5.5 shows that PDSYEVX transmits $7\, n^2/\sqrt{p}\, \log(\sqrt{p})$ words per node.

5.7 Deriving the PDSYEVX order n imbalance and overhead term on the Intel Paragon

Table 5.6 shows the origin of the $\theta(n)$ load imbalance cost on the Intel Paragon.

Table 5.6: $\theta(n)$ load imbalance cost on the `PARAGON`

Task	Full model	Paragon model	Paragon time
load imbalance during eigendecomposition (See section 4.3)	$620\gamma_1 + 112\gamma_\div$	620×0.0740 $+112\times3.85$	$477\, n$
order n overhead term in reduction to tridiagonal form (See table 4.7)	$9\delta_4 + 6\delta_2$	$9\times239+6\times23.5$	$2256\, n$
order n overhead term in back transformation (See table 4.10)	$2\,\delta_2$	2×23.5	$47\, n$
Total			$2780\, n$

Table 5.7: Order $\frac{n^2}{\sqrt{p}}$ load imbalance and overhead term on the `PARAGON`

Task	Full model	Paragon model	Paragon time
Order $n^2/\sqrt{p}$ overhead term in reduction to tridiagonal form (See table 4.7)	$2\frac{n^2}{\mathsf{nb}\,\mathsf{pbf}\,p_c}\delta_2 +2\frac{n^2}{\mathsf{nb}^2\,\mathsf{pbf}\,p_c}\delta_3+\frac{n^2}{p_c}\delta_1$	$\left(\frac{2\times23.5}{32\times2}+\frac{2\times103}{32\times32\times2} +3.97\right)\frac{n^2}{\sqrt{p}}$	$4.70\,\frac{n^2}{\sqrt{p}}$
Order $n^2/\sqrt{p}$ load imbalance term in reduction to tridiagonal form (See table 4.7)	$\frac{7}{2}\frac{n^2\,\mathsf{nb}}{p_r}\gamma_2+\frac{1}{2}\frac{n^2\,\mathsf{nb}}{p_c}\gamma_2 +\frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{p_r}\gamma_2+\frac{n^2\,\mathsf{nb}}{p_r}\gamma_3+ \frac{1}{2}\frac{n^2\,\mathsf{nb}}{p_c}\gamma_3+\frac{1}{2}\frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{p_r}\gamma_3$	$(\frac{7}{2}\times32+\frac{1}{2}\times32 +32\times2)\times0.0247 +(\frac{1}{2}\times32+\frac{1}{2}\times32 +2\times32)\times0.0215$	$6.81\,\frac{n^2}{\sqrt{p}}$
Order $n^2/\sqrt{p}$ load imbalance term in back transformation (See table 4.10)	$0.5\,\frac{n^2\,\mathsf{nb}}{p_r}\gamma_2+3\frac{n\,m\,\mathsf{nb}}{p_c}\gamma_3 +2\frac{n^2\,\mathsf{vnb}}{p_c}\gamma_3$	$(0.5\times32\times0.0247 +3\times32\times0.0215 +2\times0.0215\times0)\frac{n^2}{\sqrt{p}}$	$2.46\,\frac{n^2}{\sqrt{p}}$
Total			$14.0\,\frac{n^2}{\sqrt{p}}$

5.8 Deriving the `PDSYEVX` order $\frac{n^2}{\sqrt{p}}$ imbalance and overhead term on the Intel Paragon

The order $\frac{n^2}{\sqrt{p}}$ load imbalance and overhead term on the Intel Paragon, ?/.0 $\frac{n^2}{\sqrt{p}}$ is shown in table 5.7

See section 5.2 for details on the assumptions made to simplify the full model to the six term model. Note that vnb is assumed to be zero and that pbf is assumed to be 2.

Chapter 6

Perfomance on distributed memory computers

6.1 Performance requirements of distributed memory computers for running PDSYEVX efficiently

The most important feature of a parallel computer is its peak flop rate. Indeed, everything else is measured against the peak flop rate. The second most important feature is main memory, but which feature of main memory is most important depends on whether you want peak efficiency (i.e. using as few processors as possible) or minimum execution time (i.e. using more processors). If you plan to use only as many processors as necessary, filling each processor's memory completely, then main memory size is the most important factor controlling efficiency. If you plan to use more processors, main memory random access time becomes the most important factor.

Network performance of today's distributed memory computers is good enough to keep communication cost from being the limiting factor on performance. Furthermore, if the network performance (either latency or bandwidth) were the limiting factor, there are ways that we could reduce the communication cost by as much as $\log(\sqrt{p})$[107]. Still, if one has a network of workstations connected by a single ethernet or FDDI ring, the very low bisection bandwidth will always keep efficiency low See section 8.4.2 for details.

6.1.1 Bandwidth rule of thumb

Bandwidth rule of thumb: Bisection bandwidth per processor[1] *times the square root of memory size per processor should exceed floating point performance per processor.*

$$\frac{\text{Megabytes/sec}}{\text{processor}} \times \frac{\sqrt{\text{Megabytes}}}{\text{processor}} > \frac{\text{Megaflops/sec}}{\text{processor}}$$

assures that bandwidth will not limit performance.

The bandwidth rule of thumb shows that if memory size grows as fast as peak floating point execution rate, the network bisection bandwidth need only grow as the square root of the peak floating point execution rate. This is very encouraging for the future of parallel computing. This rule also shows that the bandwidth requirement grows as the problem sizes decreases. This rule does not make as wide a claim as the memory rule of thumb, it does not promise that `PDSYEVX` will be efficient, only that bandwidth will not be the limiting factor.

Provided the bandwidth rule of thumb holds, execution time attributable to message volume will not exceed 40% of the time devoted to floating point execution in `PDSYEVX` on problems that nearly fill memory.

6.1.2 Memory size rule of thumb

Memory size rule of thumb: memory size should match floating point performance

$$\frac{\text{Megabytes}}{\text{processor}} > \frac{\text{Megaflops/sec}}{\text{processor}}$$

assures that `PDSYEVX` will be efficient on large problems.

This rule is sufficient because it holds even if message latency and software overhead hold constant as peak performance increases and network bisection bandwidth and `BLAS2` performance increase as slowly as the square root of the increase in the peak flop rate.

[1] Bisection bandwidth per processor is the total bisection bandwidth of the network divided by the number of processors.

$$
\begin{aligned}
\frac{\text{message transmission time}}{\text{floating point execution time}} &= \frac{7.5 n^2/\sqrt{p}\,\lceil \log_2(\sqrt{p})\rceil\,\beta}{10/3\; n^3/p\,\gamma_3} && \text{Table 5.1} \\
&= \frac{7.5\,\lceil \log_2(\sqrt{p})\rceil\,\beta}{10/3\; n/\sqrt{p}\,\gamma_3} && \text{Cancel } n^2/\sqrt{p} \\
&= \frac{7.5\,\lceil \log_2(\sqrt{p})\rceil\,\beta}{10/3\;\sqrt{M\,10^6/(6\times 8)}\,\gamma_3} && n/\sqrt{p} = \sqrt{M\,10^6/(6\times 8)} \\
& && \texttt{PDSYEVX} \text{ uses } 6\,n^2/p \text{ DP words} \\
&= \frac{7.5 \times 3 \times 8 \cdot 10^{-6}/mbs}{10/3\;\sqrt{M\,10^6/(6\times 8)}\,10^{-6}/mfs} && \beta = 8\cdot 10^{-6}/mbs \\
& && \gamma_3 = 10^{-6}/mfs \\
& && \lceil \log_2(\sqrt{p})\rceil = 3 \\
&= \frac{7.5 \times 8\,\sqrt{6\times 8}\,mfs}{10/3\;10^3\,mbs} && \text{Simplify} \\
&= .374\frac{mfs}{\sqrt{M}\,mbs} && .374 = \frac{7.5\times 3\times 8\,\sqrt{6\times 8}}{10/3\;10^3}
\end{aligned}
$$

Figure 6.1: Relative cost of message volume as a function of the ratio between peak floating point execution rate in Megaflops, mfs, and the product of main memory size in Megabytes, M and network bisection bandwidth in Megabytes/sec, mbs.

Message latency and software overhead are limited by main memory access time, which decreases slowly, but bisection bandwidth and BLAS2 performance (which is limited by main memory bandwidth) continue to improve though not as rapidly as peak performance.

When the number of megabytes of main memory equals the peak floating point rate (in megaflops/sec), message latency will typically account for ten times less execution time than the time devoted to floating point execution in PDSYEVX on problems that nearly fill memory. The arithmetic in figure 6.2 justifies this statement provided that message latency does not exceed 100 microseconds.

The memory rule of thumb is too simple to capture all aspects of any computer, nonetheless we have found it to be useful. The derivation in figure 6.2 makes two main assumptions: latency is around 100 microseconds and $\lceil \log_2(\sqrt{p})\rceil = 3$. Selcom will either be exactly correct, but in our experience neither will tend to be small by more than a factor of 2 (i.e. $p\ leq 4096$). The memory rule of thumb also depends on sufficient bandwidth and on reasonable BLAS2 and software overhead costs. As we will show next, network bandwidth capacity and BLAS2 performance need not grow rapidly to support this rule and software overhead costs need only remain constant.

The memory rule of thumb holds for all computers marketed as distributed memory

$$
\begin{aligned}
\frac{\text{message latency time}}{\text{floating point execution time}} &= \frac{17\,n\,\lceil \log_2(\sqrt{p})\rceil\,\alpha}{10/3\,n^3/p\,\gamma_3} && \text{Table 5.1} \\
&= \frac{17\,\lceil \log_2(\sqrt{p})\rceil\,\alpha}{10/3\,n^2/p\,\gamma_3} && \text{Cancel } n \\
&= \frac{17\,\lceil \log_2(\sqrt{p})\rceil\,\alpha}{10/3 \times (M\,10^6/48) \times \gamma_3} && n/\sqrt{p} = \sqrt{M\,10^6/(6 \times 8)} \quad \texttt{PDSYEVX} \text{ uses } 6\,n^2/p \text{ DP words} \\
&= \frac{17 \times 3 \times 100\,10^{-6}}{10/3 \times (M\,10^6/48) \times (10^{-6}/mfs)} && \alpha = 100\,10^{-6} \quad \lceil \log_2(\sqrt{p})\rceil = 3 \quad \gamma_3 = 10^{-6}/mfs \\
&= 0.073\frac{mfs}{M} && \frac{17 \times 3 \times 100 \times 10^{-6}}{10/3/48} = 0.073
\end{aligned}
$$

Figure 6.2: Relative cost of message latency as a function of the ratio between peak floating point execution rate in Megaflops, mfs, and main memory size in Megabytes, M.

computers, but does not hold for non-scalable or extremely low bandwidth networks. One could design a distributed memory computer for which this rule does not hold, but the features that are necessary for this rule to hold are also important for a range of other applications and hence we expect this rule to hold for essentially all distributed memory computers.

The memory rule of thumb while sufficient is not necessary. It is possible to achieve efficiency on `PDSYEVX` on computers whose memory is smaller than that suggested by this rule[2]. In section 6.1.3 I discuss what properties a computer must have to allow efficient execution on smaller problem sizes.

Though meeting the memory rule of thumb is not necessary to achieve high performance, there are reasons to believe that it will be useful for several years. Software latencies are not decreasing rapidly. Software overhead, since it is tied to main memory latency, is not decreasing rapidly either. Bisection bandwidth and `BLAS2` performance is increasing, but not as fast as peak floating point efficiency.

On the other hand, improvements to `PDSYEVX` will make it possible to achieve high performance with less memory and may someday obsolete the memory rule of thumb.

[2]The `PARAGON` is an example.

6.1.3 Performance requirements for minimum execution time

If you intend to use as many processors as possible to minimize execution time, the second most important machine characteristic (after peak floating point rate) is main memory speed. Main memory speed affects three of the four sources of inefficiencies in PDSYEVX: message initiation, load imbalance and software overhead. Message initiation and software overhead costs are controlled by how long it takes to execute a stream of code with little data or code locality. Since the communication software initiation code offers little code or data locality, its execution time is largely dependent on main memory latency. Load imbalance consists mainly of BLAS2 row and column operations. The BLAS2 flop rate is controlled by main memory bandwidth. Smaller main memory bandwidth also requires a larger blocking factor in order to achieve peak floating point performance in matrix matrix multiply. Larger blocking factors mean more BLAS2 row and column operations. Hence reduced main memory speed has a double effect on the cost of row and column operations: increasing the number of them while increasing the cost per operation.

Caches can be used to improve memory performance, however the value of caches is reduced by several factors: The inner loop in reduction to tridiagonal form, the source of most of the inefficiency in PDSYEVX, is substantial and includes many subroutine calls. ScaLAPACK is a layered library which includes the PBLAS, BLAS, BLACS and the underlying communication software. The inner loop in reduction to tridiagonal form touches every element in the unreduced (trailing) part of the matrix. The second level cache is typically shared between code and data. Even the way that BLAS routines are typically coded impacts the value of caches in PDSYEVX. The fact that the inner loop in reduction to tridiagonal form includes many subroutine calls combined with ScaLAPACK's layered approach means that this inner loop typically involves many code cache misses. Indeed even the much simpler inner loop in LU involves many code cache misses in ScaLAPACK[160]. Since this same inner loop touches every element in the matrix, the secondary cache, typically shared by both code and data, will be completely flushed each time through the loop meaning that code cache misses will have to be satisfied by main memory.

The way that BLAS routines are typically optimized leads to a high code cache miss rate. BLAS routines are typically coded and optimized by timing them on a representative set of requests[92]. Each request however is typically run many times and the times are averaged. Each run may involved different data to ensure that the times represent the cost of moving the data from main memory. However, no effort is made[3] to account for the cost of moving the code from main memory. Hence, the code cache is a resource to which no cost is assigned during optimization. Loop unrolling can vastly expand the code cache requirements but it can also improve performance, at least if the code is in cache. Hence it is likely that in optimizing BLAS codes, some loops get unrolled to the point where they use half or more of the code cache. If two such codes are called in the same loop, code cache misses are inevitable. The unfortunate aspect of this is that the hardware designer is powerless to prevent it. Increasing the size of the code cache might lead to even more loop unrolling and even worse performance.

There are two ways that hardware manufacturers could make caches more useful. One would be to improve the way that BLAS codes are optimized to ensure that the code cache is a recognized resource (either by measuring code cache use in each call or by having the codes optimized on a system with smaller cache sizes than those offered to the public). The second would be to allow a path from main memory to the register file that bypasses the cache. In the inner loop of reduction to tridiagonal form, every element of the matrix is touched, but there is no temporal locality and no point in moving these elements up the cache hierarchy. If these calls to the BLAS matrix-vector multiply routine, DGEMV, could be made to bypass the caches, these caches would remain useful in the other portions of the code: i.e. software overhead and communication latency. Even row and column operations would benefit because these operations involve data locality across loop iterations, this data locality is made worthless by the fact that the loop touches every element in the matrix each time through but could be useful if certain DGEMV calls could be made to bypass the caches. This would require a coordinated software and hardware effort.

Secondary caches are of little importance in determining PDSYEVX execution time because the inner loop traverses the entire matrix without any data temporal locality within the loop. Secondary caches are important to achieving peak matrix-matrix multiply performance, but that is their only use in PDSYEVX. This is because in principle if the secondary cache were large enough and the problem small enough, secondary cache could hold the entire matrix and hence act as fast main memory. Unfortunately, secondary caches are never large enough to support an efficient problem size.

I would hope that, if there are other applications like PDSYEVX that could make

[3]It is difficult to account for the cost of moving the code from main memory.

efficient use of smaller faster memories, some vendor or vendors will build some machines with smaller faster main memory. I suspect that more applications need large slow memory, than small fast memory. Indeed, PDSYEVX, can work well either way. But, especially with improvements to PDSYEVX that will allow it to achieve high performance on smaller problem sizes, PDSYEVX could achieve impressive results on a distributed memory machine with half the main memory now typical of distributed memory parallel computers if that smaller main memory could be made modestly, say 20%, faster. With the out-of-core symmetric eigensolver being developed by Ed D'Azevedo (based on my suggestion to reduce main memory requirements from $4n^2$ to $\frac{1}{2}n^2$ by using symmetric packed storage during the reduction to tridiagonal form and two passes trhough back transformation), the main memory requirements of PDSYEVX will drop by a factor of 6 to 12, furthering the argument for smaller, faster main memory.

As ScaLAPACK improves, it will be able to achieve high efficiency on smaller problem sizes. This will mean that the best machines for ScaLAPACK will have less memory than that suggested by the memory rule of thumb at the top of this chapter.

6.1.4 Gang scheduling

6.2 sec:gang

A code which involves frequent synchronizations, such as reduction to tridiagonal form, requires either dedicated use of the the nodes upon which it runs or gang scheduling. If even one node is not participating in the computation, the computation will stall at the next synchronization point.

6.2.1 Consistent performance on all nodes

A statically load balanced code, such as PDSYEVX, will executed only as fast as the slowest node on which it is run. This, like the need for gang scheduling, is obvious. Yet, occasionally nodes which have identical specifications perform differently. Kathy Yelick noticed that some nodes CM5 at Berkeley were slower than others.. And, I have reason to believe that at least two of the nodes on the PARAGON at Univeristy of Tennessee at Knoxville are slower than the others (See Table 6.3).

The people who design and maintain distributed memory parallel computers should

Table 6.1: Performance

	message latency α	transmission cost per word β	BLAS1 flop rate γ_1	matrix-vector mutiply software overhead δ_2	matrix-vector mutiply flop rate γ_2	matrix-matrix mutiply flop rate γ_3	divide $\gamma_\div$
IBM SP2	54	0.12 (67)	.0037 270	.25 (4)	5	??	P
PARAGON	66	0.14 (57)	0.0235 (42)	3.8 (.26)	80	P	??

make sure that slow nodes are identified and marked as such or taken off-line.

6.3 Performance characteristics of distributed memory computers

6.3.1 PDSYEVX execution time (predicted and actual)

Table 6.3 compares predicted and actual performance on the Intel PARAGON. Actual PDSYEVX performance never exceeds the performance predicted by our model and usually is within 15% of the predicted performance. Every run which shows actual execution time which is more than 15% greater than expected execution time is marked with an asterisk. I would be satisfied with a performance model that is within 20% to 25%, and would not expect this performance model to match to within 15% on other machines. I have checked several of these and have noticed that in these runs one or two processors have noticeably slower performance on DGEMV than the other processors. I have also rerun many of these aberrant timings and for each that I have rerun, at least one of the runs completed within 15% of predicted performance. Nonetheless, this aberrant behavior deserves further study.

	PARAGON MP	IBM SP2[4]
Processor	50 Mhz i860 XP	120 Mhz POWER2 SC
Location	xps5.ccs.ornl.gov	chowder.ccs.utk.edu
Data cache	16K bytes 4way set-associated write-back 32-byte lines[5]	128K bytes
Code cache	16 Kbytes 4way set-associated 32-byte blocks	32K bytes
Second level cache	None	None
Processors per node	1	1
Memory per node	32 Mbytes	256 Mbytes
Operating system	Paragon OSF/1 xps5 1.0.4 R1_4_5	AIX
ScaLAPACK	1.5	1.5
BLAS	-lkmath	-lesslp2
BLACS	NX BLACS	MPL BLACS
Communication software	NX	MPI
Precision	Double 64 bits	Double 64 bits

Table 6.2: Hardware and software characteristics of the PARAGON and the IBM SP2.

Table 6.3: Predicted and actual execution times of PDSYEVX on xps5, an Intel PARAGON. Problem sizes which resulted in execution time of greater than 15% greater than predicted are marked with an asterix. Many of these problem sizes which result in more than 15% greater execution time than expected were repeated to show that the unusually large execution times are aberrant.

n	nprow	npcol	nb	Actual time (seconds)	Estimated time (seconds)	$\frac{\text{Actual}}{\text{Estimated}}$
375	2	4	32	8.51	8.24	0.97
375	4	8	32	6.34	4.65	0.73*
750	2	4	32	31.2	30.1	0.96
750	2	4	32	31.3	30.1	0.96
750	2	4	32	31.5	30.1	0.96
750	2	4	32	41.2	30.1	0.73*
750	2	4	32	43.3	30.1	0.7*
750	4	4	32	20.3	18.9	0.93
750	4	6	32	16.5	15.3	0.93
750	4	6	32	22.3	15.3	0.69*
750	4	6	32	23.1	15.3	0.66*
750	4	8	32	14.1	13.2	0.93
1000	2	4	32	55.8	53.8	0.96
1000	2	4	8	52.9	54.4	1
1000	4	2	32	56.5	54.9	0.97
1000	4	2	8	56.2	59.3	1.1
1125	2	4	32	72.2	68.8	0.95
1125	4	8	32	38.2	26.7	0.7*
1500	2	4	32	133	127	0.95
1500	2	4	32	134	127	0.95
1500	2	4	32	134	127	0.95
1500	2	4	32	176	127	0.73*
1500	2	4	32	183	127	0.7*
1500	4	4	32	77.2	72.9	0.94
1500	4	6	32	77	55	0.71*
1500	4	6	32	59.3	55	0.93
1500	4	6	32	80.9	55	0.68*
1500	4	8	32	48.6	45.2	0.93
1875	4	8	32	99.7	70.9	0.71*
2250	4	4	32	186	175	0.94
2250	4	6	32	138	127	0.92
2250	4	6	32	179	127	0.71*
2250	4	6	32	182	127	0.7*
2250	4	8	32	112	102	0.91
2625	4	8	32	203	144	0.71*
3000	4	8	32	214	191	0.89

Chapter 7

Execution time of other dense symmetric eigensolvers

In this chapter, I present models for performance of other symmetric eigensolvers. These models have not been fully validated, although some have been partly validated.

7.1 Implementations based on reduction to tridiagonal form

7.1.1 PeIGs

PeIGs[74], like PDSYEVX, uses reduction to tridiagonal form, bisection, inverse iteration and back transformation to perform the parall eigendecomposition of a dense symmetric matrix. The execution time of PeIGs differs from that of PDSYEVX for two significant reasons: PeIGs is coded differently, (using a different language and different libraries) than PDSYEVX and it uses a different re-orthogonalization strategy. I am more interested in the difference resulting from the different re-orthogonalization strategy.

In PDSYEVX the number of flops performed by any particular processor, p_i, during re-orthogonalization is: $\sum_{C \in \{\text{clusters assigned to } p_i\}} 4 \sum_{i=1}^{\text{size}(C)} \mathsf{n\_iter}(i)\, n\,(i-1)$. Where: $\mathsf{n\_iter}(i)$ is the number of inverse iterations performed for eigenvalue i (typically 3). If the size of the largest cluster is greater than $\frac{n}{p}$, the processor which is responsible for this cluster will not be responsible for any eigenvalues outside of this cluster.

Hence, if the size of the largest cluster is greater than $\frac{n}{p}$, the number of flops performed by the processor to which this processor is assigned is (on average):

$$4 \, \mathsf{n\_iter} \, n \, \frac{1}{2} c^2 = 6 n \, c^2$$

where: $c = \max_{C \in \{\text{clusters}\}} \text{size}(C)$ i.e. the number of eigenvalues in the largest cluster, and $\mathsf{n\_iter} = 3$ is the average number of inverse iterations performed for each eigenvalue.

If the largest cluster has fewer than $\frac{n}{p}$ eigenvalues, the number of eigenvalues that will be assigned to any one processor, and hence the total number of flops it must perform, is limited. The worst case is where there are $p+1$ clusters each of size $\frac{n}{p+1}$. In this case, one processor must be assigned 2 clusters of size $\frac{n}{p+1}$, requiring (on average) $2 \times 6 \, n \, (\frac{n}{p+1})^2$ or roughly $12 \frac{n^3}{p^2}$ flops.

In contrast, `PeIGs` uses multiple processors and simultaneous iteration to maintain orthogonality among eigenvectors associated with clustered eigenvalues. Traditional inverse iteration[102] computes one eigenvector at a time, re-orthogonalizing against all previous eigenvectors associated with eigenvalues in the same cluster, after each iteration. `PeIGs`, in what they refer to as simultaneous iteration, performs one step of inverse iteration on all eigenvectors associated with a cluster of eigenvalues and then reorthogonalizes all the eigenvectors. This allows the re-orthogonalization to be performed efficiently in parallel.

`PeIGs` is more accurate but slower than `PDSYEVX` if the input matrix has large clusters of eigenvalues[1] The cost of re-orthogonalization in `PeIGs` is $O(n^2 \, c/p)$ flops versus $O(nc^2)$ flops in `PDSYEVX`.

7.1.2 HJS

Hendrickson, Jessup and Smith[91] wrote a symmetric eigensolver, `HJS`, for the `PARAGON` which is significantly faster than `PDSYEVX`, but which has never been released, and only works on the Intel `PARAGON`.

`HJS` requires that the data layout block size be 1, i.e. a cyclic distribution, that the processor grid be square, i.e. $p_r = p_c$ and that intermediate matrices be replicated across processor columns and distributed across processor rows. The requirement that the processor grid be square limits efficiency when used on a non-square processor gird. They show that the algorithmic block size need not be tied to the data layout block size. At the time that `PDSYTRD` was written, the `PBLAS` could not efficiently use a cyclic distribution and

[1] `PDSYEVX` can maintain orthogonality among eigenvectors associated with clusters up to $\frac{n}{p}$ eigenvalues easily and efficiently.

did not support matrices replicated in one processor dimension and distributed across the other.

HJS has several advantages over PDSYEVX. It uses a more efficient transpose operation, eliminates redundant communication, reduces the number of messages by combining some and reduces the number of words transmitted per process by using recursive halving and doubling. HJS also reduces the load imbalance by a factor of $\sqrt{p}$ by using a cyclic data layout and using all processors in all calculations[2]. ScaLAPACK will incorporate several of these ideas into the next version of PDSYEVX.

HJS notation

HJS also differs in a couple other rather minor aspects. They compute the norm of v in a manner which could overflow, and they represent the reflector in a manner could likewise overflow. These reduce execution time and program complexity slightly.

Their manner of counting the cost of messages in their performance model differs from ours also. They count the cost of a message swap (sending a message to and simultaneously receiving a message from another processor) as equal to cost of sending a single message. This reflects reality on the PARAGON and many but not all distributed memory machines. Using their method would not significantly change the model for PDSYEVX because PDSYEVX does not use message swap operations.

In their paper[91], they use different variable names for the result of each computation, and show all indices explicitly. Figure 7.1 relates their notation to ours.

Figure 7.1: HJS notation

HJS	our equivalent	details
L	tril(A)	
x_α	w	tril$(A)v$
y_α	w^T	tril$(A,-1)v^T$
p	w	w + transpose w^T
γ	c	not mathematically identical

[2] PDSYTRD uses only p_r processors in many computations

7.1.3 Comparing the execution time of HJS to PDSYEVX

The HJS implementation of parallel blocked Household tri-diagonalization performs essentially the same computation as PDSYEVX. The difference is in the communication, load balance and overhead costs. However, the operations are not performed in the same order, and hence the steps don't match exactly. Some of the costs, particularly communication costs, could easily have been assigned to a different operation than the one that I assigned them to. Hence, the execution time models for each of the individual tasks should not be taken in isolation but understood as an aid in understanding the total.

Updating the current column of A (Line 1.1 in Figure 7.2)

As shown in table 4.1, the cost of updating the current column of A in PDSYTRD is:

$$2n\lceil\log_2(\sqrt{p})\rceil\,\alpha + n\,\mathsf{nb}\,\lceil\log_2(\sqrt{p})\rceil\beta + 2\,n\,\delta_2 + \frac{n^2\,\mathsf{nb}}{\sqrt{p}}\,\gamma_2 + 2n\,\delta_4$$

In Figure 6[91] steps Y2, 10.1, 10.2 and 10.3 of HJS are involved in updating the current column of A and the cost of these steps is:

$$n\,\alpha + \frac{1}{2}\frac{n^2}{\sqrt{p}}\,\beta + 2\,n\,\delta_2 + \frac{n^2\,\mathsf{nb}}{p}\,\gamma_2\ .$$

In PDSYEVX, a small part of v^T and w^T must be broadcast within the current column of processors. In HJS, there is no need to broadcast v^T because it is already replicated across all processor rows. Instead of broadcasting the piece of w^T that is necessary for this update, HJS transposes all of w^T, (cost: $n\,\alpha + 1/2\,n^2/\sqrt{p}\beta$) anticipating the need for this in the rank $2k$ update.

The number of DGEMV flops performed does not change, but they are distributed across all of the processors instead of being shared only by one column of processors. In order to allow these flops to be distributed across all the processors, this update is performed in a right-looking manner, i.e. the entire block column of the remaining matrix is updated with the Householder reflector. In PDSYEVX, this update is performed in a left looking manner, only the current column is updated (with a matrix vector multiply). In PDSYEVX, the right-looking variant does not spread the work any better and hence the left-looking variant is preferred because it involves a matrix-vector multiply, DGEMV, rather than a rank-one update, DGER. Matrix-vector multiply requires only that every matrix element be read. A rank-one update requires that every matrix element be read and then re-written.

The δ_4 term does not exist for HJS because they do not use the PBLAS, avoiding the error checking and overhead associated with the PBLAS.

Computing the reflector (Line 2.1 in Figure 7.2)

As shown in table 4.2, the cost in PDSYTRD is: $3\,n\,\lceil \log_2(p_r)\rceil\,\alpha + n\,\delta_4$.
In Figure 6[91] steps 2, 3, 4, 5, 6 and X of HJS are involved in computing the reflector, and the cost of these steps is: $n\,\lceil \log_2(p)\rceil\,\alpha$, a little less than the cost in PDSYTRD.

Step 1 in HJS is also used in the computation of the reflector in HJS, however step 1 isn't necessary to compute the reflector, and it is necessary for the matrix-vector multiply, hence I assign the cost of Step 1 to the matrix-vector multiply.

Both routines perform essentially the same operations. HJS appends the broadcast of $A(J+1,J)$ to the computation of $xnorm$ (though HJS actually computes $xnorm^2$), which HJS performs as a sum-to-all. On the other hand, they involve all processors rather than just one column of processors, hence the sum costs $\lceil \log_2(p)\rceil$ rather than $\lceil \log_2(p_r)\rceil$.

The difference in performance would appear more dramatic if I included the cost of the BLAS1 operations in my PDSYEVX model. I do not because they account for an insignificant $O(\frac{n^2}{p_r})\gamma_1$ execution time. HJS performs fewer BLAS1 flops (because they do not go the extremes that PDSYTRD does to avoid overflow) and the flops that they perform are distributed over all processors instead of over only one column of processors.

The cost of matrix vector multiply(Lines 3.1-3.6 in Figure 7.2)

As shown in table 4.3, the cost of the matrix vector multiply in PDSYTRD is:

$$4\,n\,\lceil \log_2(\sqrt{p})\rceil\,\alpha + 2\,n\,\alpha + 2\,\frac{n^2\lceil \log_2(\sqrt{p})\rceil}{\sqrt{p}}\,\beta + 1.5\,\frac{n^2}{\sqrt{p}}\,\beta + 2\,n\,\mathsf{nb}\,\lceil \log_2(\sqrt{p})\rceil\,\beta + \frac{n^2}{\mathsf{nb}\,\sqrt{p}}\,\delta_2$$
$$+\frac{2}{3}\frac{n^3}{p}\,\gamma_2 + 3\,\frac{n^2\,\mathsf{nb}}{\sqrt{p}}\,\gamma_2 + \frac{n^2}{\sqrt{p}}\,\delta_1 + n\,\delta_4$$

In Figure 6[91] steps 1, Y1, 7.1, 7.2, and 7.3 are involved in matrix vector multiply and the cost of these steps is:

$$2\,n\,\lceil \log_2(\sqrt{p})\rceil\,\alpha + 2\,n\,\alpha + \frac{1}{2}\frac{n^2}{\sqrt{p}}\,\lceil \log_2(\sqrt{p})\rceil\,\beta + \frac{3}{2}\frac{n^2}{\sqrt{p}}\beta + 2\,n\,\delta_2 + \frac{1}{3}\frac{n^3}{p}\,\gamma_2\,.$$

The model for HJS is much simpler because 1) the local portion of the matrix-vector multiply requires just a single call to DGEMV and 2) the load imbalance in HJS is negligible ($O(\frac{n^2}{p})$ versus $O(\frac{n^2}{\sqrt{p}})$) in PDSYEVX).

The communication performed in HJS during the matrix vector multiply includes:

	Figure 6[91]	Execution time model
Broadcast v within a row	Step 1)	$n\,\lceil \log_2(\sqrt{p})\rceil\,\alpha+\frac{1}{2}\frac{n^2}{\sqrt{p}}\,\lceil \log_2(\sqrt{p})\rceil\,\beta$
Transpose v and y	Steps Y1, 7.3	$2\,n\,\alpha+\frac{n^2}{\sqrt{p}}\beta$
Recursive halve p	Step 7.3	$n\,\lceil \log_2(\sqrt{p})\rceil\,\alpha+\frac{1}{2}\frac{n^2}{\sqrt{p}}\beta$

The transpose operations take advantage of the fact that $p_r = p_c$. Each processor (a, b) simply sends its local portion of the vector to processor (b, a) while receiving the transpose from that same processor.

The recursive halving operation is a distributed sum in which each of the p_c processors in the row starts with k values and end up with $\frac{k}{p_c}$ sums.

Updating the matrix vector multiply (Line 4.1 in Figure 7.2)

As shown in table 4.4, the cost of updating the matrix vector multiply in PDSYTRD is:

$$6\,n\,\lceil \log_2(\sqrt{p})\rceil\,\alpha+\frac{n^2\lceil \log_2(\sqrt{p})\rceil}{p_r}\,\beta+3\,n\,\mathsf{nb}\,\lceil \log_2(\sqrt{p})\rceil\,\beta+4\,n\,\delta_2+2\,\frac{n^2\,\mathsf{nb}}{\sqrt{p}}\,\gamma_2+4\,n\,\delta_4\;.$$

In Figure 6[91] step 7.4 updates the matrix vector multiply and the cost of this step is:

$$2\,n\,\delta_2+\frac{n^2\,\mathsf{nb}}{p}\,\gamma_2+\;.$$

Computing the companion update vector, w (Line 5.1 in Figure 7.2)

As shown in table 4.5, the cost of computing the companion update vector in PDSYTRD is:

$$2\,n\,\lceil \log_2(\sqrt{p})\rceil\alpha+n\,\delta_4+\;.$$

In Figure 6[91] steps 8 and 9 compute the companion update vector and the cost of these steps is:

$$5\,n\,\lceil \log_2(\sqrt{p})\rceil\,\alpha+\frac{n^2}{\sqrt{p}}\,\beta\;.$$

Just as in the computation of the reflector, the $O(n^2)$ costs of the BLAS1 operations is insignificant. HJS performs these more efficiently than PDSYEVX, because it uses all the processors in these computations.

Perform the rank $2k$ update(Line 6.3 in Figure 7.2)

As shown in table 4.6, the cost of the rank $2k$ update in PDSYTRD is:

$$4\,\frac{n}{\mathsf{nb}}\,\lceil\log_2(\sqrt{p})\rceil\,\alpha + 2\,\frac{n^2}{\sqrt{p}}\lceil\log_2(\sqrt{p})\rceil\,\beta + \frac{n^2}{\sqrt{p}}\,\beta - 2\,n\,\mathsf{nb}\,\lceil\log_2(\sqrt{p})\rceil\,\beta$$
$$+4\,\frac{n^2}{\mathsf{nb}^2\,\sqrt{p}\,\mathsf{pbf}}\,\delta_3 + \frac{2}{3}\,\gamma_3 + 3\,\frac{n^2\,\mathsf{nb}}{\sqrt{p}}\,\gamma_3\,.$$

In Figure 6[91] step 10.4 performs the rank $2k$ update and the cost of this step is:

$$2\,\frac{n^2}{\mathsf{nb}^2\,\sqrt{p}}\,\delta_3 + \frac{2}{3}\,\gamma_3 +\,.$$

HJS does not require any communication here because W and V, are already replicated across the processor rows, while WT and VT are already replicated across all the processor columns.

Both HJS and PDSYEVX must perform the rank 2k update as a series of panel updates using DGEMM. Both PDSYTRD and HJS use a panel width of twice the algorithmic blocking factor.

Figure 7.2 summarizes the main sources of inefficiencies in HJS reduction to tridiagonal form.

Table 7.1 compares the execution time in PDSYEVX and HJS reduction to tridiagonal form. Each row represents a particular operation. The second column is the time (in seconds) associated with the given operation in PDSYEVX. The third column shows the number of the given operation performed in PDSYEVX. The product of the third column with the first column, after substituting the cost given for the operation given in section 5.2 and $n = 4000$ and $p = 64$ is the second column. For example the cost of matrix multiply flops in PDSYTRD on the PARAGON is: $2/3\,(n = 4000)^3/(p = 64)\,(\gamma_3 = .0215e-6) = 14.3$. Likewise, the second to last column (the number of the given operation performed in reduction to tridiagonal form in HJS) times the first column equals the last column (the time associated with the given operation in reduction to tridiagonal form in HJS.)

Columns 4 through 10 represent unimplemented intermediate variations on reduction to tridiagonal form. Column 4, labeled "minus PBLAS inefficiencies" assumes that a couple inefficiencies of the PBLAS are removed: (a bug in the PBLAS causing unnecessary communication and the PBLAS overhead). Column 5, labeled "be less paranoid", assumes that in addition PDSYTRD computes reflectors in the slightly faster, slightly riskier manner

Figure 7.2: Execution time model for HJS reduction to tridiagonal form. Line numbers match Figure 4.5(PDSYEVX execution time)

		computation		communication	
		overhead	imbalance	latency	bandwidth
	do $ii = 1, n, \mathsf{nb}$				
	$mxi = \min(ii + \mathsf{nb}, n)$				
	do $i = ii, mxi$				
Update current (i^{th}) column of A					
1.1	transpose w			$n \lg(\sqrt{p})\,\alpha$	$\frac{1}{2}\frac{n^2}{\sqrt{p}}\beta$
1.2	$A = A - W\,V^T - V\,W^T$				
Compute reflector					
2.1	$v = \text{house}(A)$			$2\,n \lg(\sqrt{p})\,\alpha$	
Perform matrix-vector multiply					
3.1	spread v across			$n \lg(\sqrt{p})\,\alpha$	$\frac{1}{2}\frac{n^2 \lg(\sqrt{p})}{\sqrt{p}}\beta$
3.2	transpose v				$\frac{1}{2}\frac{n^2}{\sqrt{p}}\beta$
3.3	$w = \text{tril}(A)v$; $w^T = \text{tril}(A, -1)v^T$		$\frac{2}{3}\frac{n^3}{p}\gamma_2$		
3.5	recursive halve w				$\frac{1}{2}\frac{n^2}{\sqrt{p}}\beta$
3.6	$w = w + \text{transpose } w^T$				$\frac{1}{2}\frac{n^2}{\sqrt{p}}\beta$
Update the matrix-vector product					
4.1	$w = w - W\,V^T v - V\,W^T v$			$3\,n \lg(\sqrt{p})\,\alpha$	$\frac{n^2}{\sqrt{p}}\beta$
Compute companion update vector					
5.1	$c = w \cdot v^T$; $w = \tau\,w - (c\,\tau/2)\,v$			$2n \lg(\sqrt{p})\,\alpha$	
	end do $i = ii, mxi$				
Perform rank $2k$ update					
6.3	$A = A - W\,V^T - V\,W^T$	$2\,\frac{n^2}{\mathsf{nb}^2\sqrt{p}}\delta_3$	$\frac{2}{3}\frac{n^3}{p}\gamma_3$		
	end do $ii = 1, n, \mathsf{nb}$				

that HJS does. Column 6 assumes direct transpose operations. Column 7 assumes that certain messages are combined, reducing the message latency cost. Column 8 assumes that sum-to-all is used instead of sum-to-one follow by a broadcast, reducing the latency cost. Column 9, assumes that V, W, VT, WT are stored replicated across processor columns, this eliminates all communication in the rank 2k update. Storing the data replicated also allows all processors to be involved in all computations, but this is not assumed until column 11. Column 10 assumes a cyclic data layout, eliminating some load imbalance. Column 11 assumes that all processors are involved in all computations, eliminating the load imbalance which was not eliminated by using a cyclic data layout.

7.1.4 PDSYEV

`PDSYEV` uses the QR algorithm to solve the tridiagonal eigenproblem. Each eigenvector is spread evenly among all the processors. Each processor redundantly computes the rotations and updates the portion of each eigenvector which it owns. Computing the rotations requires $O(n^2)$ flops, whereas updating the eigenvectors requires $O(n^3)$ flops. Hence `PDSYEV` scales reasonably well as long as all the eigenvectors are required.

Each rotation requires 2 divides, 1 square root and approximately 20 to compute and 6 flops to apply.
The cost of the QR based tridiagonal eigensolution in `PDSYEV` is:

$$\sum_{j=1}^{n} \mathsf{sweeps}(j)\,(n-j)\,(2\,\gamma_{\div} + \gamma_{\surd} + 20\,\gamma_1 + \frac{1}{p} 6\,n\,\gamma_1)$$

On average, it takes two sweeps per eigenvalue, so we set $\mathsf{sweeps}(j) = 2$ and simplify:

$$2\,n^2\gamma_{\div} + 1\,n^2\gamma_{\surd} + 20\,n^2\gamma_1 + 6\,\frac{m\,n^2}{p}\gamma_1$$

7.2 Other techniques

7.2.1 One dimensional data layouts

One dimensional data layouts can improve the performance of dense linear algebra codes on modest numbers of processors, especially on one-sided reductions like LU and QR decomposition. In general, one dimensional data layouts require fewer communication calls in the inner loop but more words transmitted per process. One-sided reductions typically require fewer messages within rows than within columns, sometimes by a factor as high as nb, other times the advantage is a more modest $\log(\sqrt{p})$. One-sided reductions often require fewer words to be transmitted between columns than between rows of processors, usually by a factor of nb.

One dimensional data layouts also offer less overhead. Often an entire block column can be computed by a call to the corresponding `LAPACK` code rather than the `ScaLAPACK` code, saving significant overhead costs.

Table 7.1: Comparison between the cost of HJS reduction to tridiagonal form and PDSYTRD on $n = 4000, p = 64, \mathsf{nb} = 32$. Values differing from previous column are shaded.

scale factor	PDSYTRD estimated time	PDSYTRD counts	minus PBLAS inefficiency	be less paranoid	direct transpose	merge operations	use sum-to-all	Store V, W, V^T, W^T replicated	no data blocking	All processors compute (i.e. HJS)	HJS estimated time
$\frac{n^3}{p}\gamma_3$	14.3	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	14.3
$\frac{n^3}{p}\gamma_2$	16.4	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	16.4
$\frac{n^2}{\sqrt{p}\,\mathsf{nb}^2\,\mathsf{pbf}}\delta_3$	2.1	2	2	2	2	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0.5
$n\,\delta_2$	0.6	6	6	6	6	5	5	5	4	4	0.4
$\frac{n^2}{\sqrt{p}\,\mathsf{nb}\,\mathsf{pbf}}\delta_2$	4.7	2	2	2	2	1	1	1	0	0	0.0
$\frac{n^2}{\sqrt{p}}\delta_1$	4.0	$\frac{1}{2}$	0	0	0	0	0	0	0	0	0.0
$\frac{n^2}{\sqrt{p}}\delta_1$	1.7	$\frac{1}{2}$	0	0	0	0	0	0	0	0	0.0
$n\,\delta_4$	8.9	$9\frac{1}{2}$	0	0	0	0	0	0	0	0	0.0
$\frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{\sqrt{p}}\gamma_2$	0.9	1	1	1	1	1	1	1	0	0	0.0
$\frac{n^2\,\mathsf{nb}}{\sqrt{p}}\gamma_2$	1.7	4	4	4	4	4	4	4	3	0	0.0
$\frac{n^2\,\mathsf{nb}\,\mathsf{pbf}}{\sqrt{p}}\gamma_3$	1.0	1	1	1	1	1	1	1	1	1	1.0
$\frac{n^2\,\mathsf{nb}}{\sqrt{p}}\gamma_3$	0.5	1	1	1	1	1	1	1	0	0	0.0
$n\lceil\log_2(\sqrt{p})\rceil\alpha$	13.5	17	15	14	12	9	6	6	6	9	7.1
$n\,\alpha$	0.5	2	2	2	4	4	4	3	3	3	0.8
$\frac{n}{\mathsf{nb}}\lceil\log_2(\sqrt{p})\rceil\alpha$	0.3	4	4	4	2	1	1	0	0	0	0.0
$\frac{n^2}{\sqrt{p}}\lceil\log_2(\sqrt{p})\rceil\beta$	4.4	5	4	4	2	2	2	$1\frac{1}{2}$	$1\frac{1}{2}$	$\frac{1}{2}$	0.8
$\frac{n^2}{\sqrt{p}}\beta$	0.6	2	2	2	2	2	2	$1\frac{1}{2}$	$1\frac{1}{2}$	$2\frac{1}{2}$	0.7
$n\,\mathsf{nb}\lceil\log_2(\sqrt{p})\rceil\beta$	0.02	8	7	7	5	5	5	$2\frac{1}{2}$	0	0	0
Total est. time		76	59	58	55	51	49	48	41	42	
Actual time		93.								61.	

Both LU decomposition and back transformation would benefit considerably from one-dimensional data layouts when p is small, although the advantage would be most pronounced on LU. One-sided reductions require $O(n)$ reductions across processor rows but only $O(\frac{n}{\mathsf{nb}})$ reductions across processor columns. On a high latency system such as a network of workstations, the performance improvement from using a one-dimensional data layout could be substantial since LU requires $O(\mathsf{nb})$ fewer messages on a one-dimensional data layout.

ScaLAPACK does not take full advantage of one-dimensional data layouts because it calls the ScaLAPACK code even when the LAPACK code would do the job faster.

Two-sided reductions, such as reduction to tridiagonal form, do not benefit from one dimensional data layouts. Two-sided reductions require $O(n)$ reductions across processor rows and $O(n)$ reductions across processor columns,hence eliminating the reductions across processor rows (by using a 1D data decomposition) will not substantially reduce the number of messages in two-sided reductions.

7.2.2 Unblocked reduction to tridiagonal form

Unblocked reduction to tridiagonal form can outperform blocked reduction for small and modest sized problems, especially if a good compiler is available for the inner kernel. Unblocked reduction to tridiagonal form must perform all of its flops as BLAS2 flops, whereas blocked reduction to tridiagonal form performs half of its flops as BLAS3 flops. However, unblocked reduction to tridiagonal form requires much less overhead. Blocked reduction to tridiagonal form requires at least $6n$ calls to DGEMV. unblocked reduction to tridiagonal form requires only n calls to DSYMV and n calls to DGER.

If a compiler is available that will efficiently compile the following kernel, unblocked reduction to tridiagonal form could require only n BLAS2 calls and still attain near peak performance on large problem sizes, especially for Hermitian eigenproblems[3]. The kernel shown below only requires each element of A be read once and written once, while performing 8 flops. This ratio, 1 memory read, 1 memory to 8 flops is one that many modern computers can handle at near peak speed, even from main memory - in part because the access are essentially all stride 1.

[3]Complex arithmetic requires only half as much memory traffic per flop

```
for i = 1, n {
    for j = 1, i {
      A(i,j) = A(i,j) - v(i) * wt(j) - w(i) * vt(j);
      nwt(i) = nwt(i) + A(i,j) * nv(j);
      nw(j) = nw(j) + A(i,j) * nvt(i);
    }
  }
```

7.2.3 Reduction to banded form

Reducing a dense matrix to banded form can be more efficient than reduction to tridiagonal form[24, 25, 116], however it is not clear that this can be made to be fast enough to overcome the added costs to the rest of the code. Reduction to banded form requires less execution time than reduction to tridiagonal form because it requires fewer messages $O(n/\mathsf{nb})$ instead of $O(n)$ and because asymptotically all of the flops can be performed as BLAS3 flops rather than half BLAS2 flops.
An efficient eigensolver based on reduction to banded form could be designed as follows:

Reduce to banded form
Reduce from banded form to tridiagonal form (do not save rotations)
Compute eigenvalues using bisection on tridiagonal form
Perform inverse iteration on banded form
Back transform the eigenvectors

This would be even simpler if only eigenvalues were required, as that eliminates the inverse iteration and back transformation steps.

If only a few eigenvectors are required, one could reduce from banded form to tridiagonal form, saving the rotations. This would allow the eigenvectors to be computed on the tridiagonal using inverse iteration (or the new Parlett/Dhillon work). Then the rotations could be applied as necessary and finally the eigenvectors would be transformed back. This would result in a complex code.

If two step band reduction to tridiagonal form were performed as above and the eigenvectors were computed on the tridiagonal matrix, the cost of transforming them back to the original problem would be at least $4n^3$, adding 60% more $O(n^3)$ flops to full tridiagonal eigendecomposition. This could be done in two steps, applying first the rotations accrued during reduction from banded to tridiagonal form and then transforming the eigenvectors of the banded form back to the original problem. A cleaner, though more costly solution would be to form the back transformation matrix after (or during) reduction to banded form, update that during reduction to banded form and then use this to transform the eigenvectors of the tridiagonal back to the original problem.

Using reduction to banded form in an eigensolver requires, at a minimum, that two step band reduction to tridiagonal form be faster than direct reduction to tridiagonal form. If eigenvectors are required, it must be significantly faster in order to overcome the additional $2n^3$ cost of back transformation.

So far, no one has demonstrated that two step reduction to tridiagonal form can be performed faster than direct reduction on distributed memory computers. Alpatov, Bischof and van de Geijn's two-step reduction to tridiagonal form[173] is not faster than `PDSYTRD`. They assert that it can be optimized, but that is also true of `PDSYTRD`. So, it is not yet clear whether two-step reduction to tridiagonal form will be significantly faster than direct reduction to tridiagonal form on any important subset of distributed memory parallel computers.

I believe that software overhead plays a significant role in limiting the performance of two step reduction to banded form.

7.2.4 One-sided reduction to tridiagonal form

Hegland et al.[90] show that one can reduce the Cholesky factor (of a shifted input matrix) to bidiagonal form updating from only one side. The result, in their implementation, is a code which requires $(10/3n^3/p + n^2 * p)$ flops per processor, $(n^2 * p)$ words communicated per processor and $(n * p)$ messages per processor.

They argue that this technique, despite requiring 2.5 times as many flops, yields better performance on their target machine than conventional methods for reduction to tridiagonal form. They use a 1D processor grid, a unblocked algorithm, a non-scalable pattern communication and computation and ignore symmetry. By ignoring much of the conventional wisdom they have achieved a simple, high performance code for their target machine (vector).

7.2.5 Strassen's matrix multiply

The number of flops in Strassen's' matrix matrix multiply is:

$$2m\, n\, k \left(\frac{\min(m,n,k)}{s_{\frac{1}{2}}}\right)^{3-\log 7}.$$

Where $s_{\frac{1}{2}}$ is the break even point for a particular Strassen's implementation, i.e. the point at which one additional Strassen's divdied and conquer step neither increases nor decreases execution time. Three factors contribute to preventing the use of Strassen's in reduction to tridiagonal form and back transformation:

$s_{\frac{1}{2}}$ **is still too large**

Lederman et al.[96] have reduced $s_{\frac{1}{2}}$ to the range 100 to 500.

k **is modest** (where k is the block size.)

We can increase the block size, but only at the cost of additional load imbalance.

$n^{3-\log 7} = n^{-.193}$ **shrinks slowly**

Increasing n by enough to improve the ratio of "Strassen flops" to standard matrix multiply flops by 50% requires a thousand-fold increase in the amount of memory required. ($5^{-.193} \approx .5$, hence n must increase by a factor of 32 to to improve the ratio of Strassen flops to standard matrix multiply flops.) Improving the ratio of "Strassen flops" to standard matrix multiply flops by increasing the number of processors involved is even difficult. Although Chou et al.[43] have shown that 7^k processors can be used to do the work of 8^k, it takes $7^5 = 16807$ processors to get a factor of two advantage this way. ($7^5/8^5 = .51$)

It is this last point that prevents Strassen's from rescuing ISDA (which is described below). Because $32^{-.193} \approx 0.5$, the problem size must be $32\, s_{\frac{1}{2}}$ in order to halve the number of flops required in ISDA. Halving the number of flops again would require that n be increased by another factor of 30, increasing memory by another factor of 900 and the total number of flops, even after the factor of two savings, by $\frac{1}{2}30^3 = 13,500$. I have not yet seen a Strassen's matrix matrix multiply that achieves twice the performance of a regular matrix matrix multiply.

Table 7.2: Fastest eigendecomposition method

	$n > 500\sqrt{p}$	$n < 500\sqrt{p}$
Random matrices	Tridiagonal ($>$ 4 times faster)	Tridiagonal
Spectrally diagonally dominant matrices	Tridiagonal	Jacobi

7.3 Jacobi

7.3.1 Jacobi versus Tridiagonal eigensolvers

This section is based on models that have only been informally validated. I have compared my models to those used by Arbenz and Slapničar[9] and Littlefield and Maschhoff[125] as well as against the execution times reported in these papers but have not performed any independent validation. Hence, the opinions that I express in this section should be taken as conjectures.

Large matrices[4] can be solved faster by a tridiagonal based eigensolver than by a Jacobi eigensolver, but it is likely that Jacobi will outperform tridiagonal based eigensolvers on small spectrally diagonally dominant matrices[5]. Since tridiagonal based methods require, asymptotically, no more than a quarter as many flops as blocked Jacobi methods, even on spectrally diagonally dominant matrices, I expect that tridiagonally based methods will win on large matrices, even spectrally diagonally dominant ones, because tridiagonal based methods can achieve 25% of peak performance on large matrices as shown in Chapter 5. I also expect that tridiagonal based eigensolvers will beat Jacobi eigensolvers on random matrices regardless of their size because on random matrices, tridiagonal eigensolvers perform roughly 16 times fewer flops[6] and I don't think that Jacobi methods will be 16 times faster per flop regardless of the input size. Table 7.2 summarizes which eigensolution method I expect to be faster as a function of these input matrix characteristics.

[4] On current machines ($n > 500\sqrt{p}$) is sufficiently large to allow a tridiagonal eigensolver to outperform Jacobi.

[5] Spectrally diagonally dominant means that the eigenvector matrix, or a permutation thereof, is diagonally dominant. Most, but not all, diagonally dominant matrices are spectrally diagonally dominant. For example if you take a dense matrix with elements randomly chosen from $[-1, 1]$ and scale the diagonal elements by $1e3$ the resulting diagonally dominant matrix will generally) be spectrally diagonally dominant. However, if you take that same matrix and add $1e3$ to each diagonal element, the eigenvector matrix is unchanged even though the matrix is clearly diagonally dominant.

[6] Assuming Jacobi converges in an optimistic 8 sweeps

7.3.2 Overview of Jacobi Methods

Despite Jacobi's simplicity there are several possible variants, especially for a parallel code, each of which have advantages. In section 7.3.16 I describe the code that I would write if I were going to write a parallel code. I recommend a 2D data layout if one wishes to be able to run efficiently on large numbers of processors (say 48 or more). However, a 1D data layout is considerably simpler to implement and simpler implementation translates into less software overhead. On some computers, Jacobi with a 1D data layout might be efficient for hundreds of processors. I recommend using a one-sided, blocked, non-threshold Jacobi[9] with a caterpillar track pairing[150] and distinct communication and computation phases, but other methods cannot be entirely rejected. For a spectrally diagonally matrix the fastest serial Jacobi algorithm is a threshold Jacobi, hence threshold methods cannot be ignored. A threshold method would almost certainly have to be two-sided, use a different pairing strategy and either a non-blocked code or some unconventional blocking strategy. Non-blocked codes may make sense for small matrices and large numbers of processors as well as for machines, such as vector architectures, which offer comparable `BLAS1` and `BLAS3` performance. Overlapping communication and computation will save time, but my experience indicates that the savings is limited.

My recommendation is weighted toward small matrices that are modestly spectrally diagonally dominant, but not so dominant that certain matrix entries can be completely ignored. If the input matrix is sparse and so strongly spectrally diagonally dominant that the matrix never fills in, one would have to consider threshold methods and methods that don't update parts of the matrix that remain zero. On the other hand, if the matrix is quite large, performance could be further improved by using a different data layout from the one that I recommend.

There are many implementation options available to anyone writing a Jacobi code. I will discuss many of these implementation options in the following sections. Section 7.3.3 explains the basic variants and data layout options. Section 7.3.4 explains the computation requirements of each of the basic variants. Section 7.3.5 explains the communication requirements of each of the basic variants. Section 7.3.6 discusses blocking (both communication and computation). Section 7.3.7 discusses the importance of exploiting symmetry. Section 7.3.8 explains that one-sided methods need not recompute diagonal blocks of $A^T A$. Section 7.3.9 discusses options for the partial eigendecomposition required by a blocked

Jacobi method. Section 7.3.10 discusses threshold strategies. Section 7.3.12 discusses pre-conditioners. Section 7.3.13 discusses overlapping communication and computation.

7.3.3 Jacobi Methods

The matlab code for the classical, two-sided, Jacobi method shown in figure 7.3 differs from textbook descriptions only in that the rotation is computed by calling `parteig` and the off diagonals are compared to the diagonals (in the threshold test) in an unusual manner. Figure 7.6 gives inefficient matlab code for `parteig` which calls matlab's eig() routine and sorts the eigenvalues to guarantee convergence. In a real implementation, `parteig` would be one or two sweeps of two-sided Jacobi.

A two-sided blocked Jacobi matlab code is given in figure 7.4. Because the code in figure 7.3 uses `parteig` to compute the rotations and norm in the threshold test, the only difference between the blocked and unblocked versions is the definition of I and J. `parteig` is not typically a full eigendecomposition, more often it is a single sweep of Jacobi.

The one-sided Jacobi variants can operate on any matrix whose left singular vectors are the same as, or related to, the eigenvectors of the input matrix. This allows many choices for pre-conditioning the input matrix, several of which are discussed in section 7.3.12

The one-sided Jacobi methods lose symmetry, but still require fewer flops than the two-sided Jacobi methods because they do not have to update the eigenvectors separately[7]. Furthermore, the one-sided Jacobi methods always access the matrix in one direction (by column for Fortran). A typical one-sided Jacobi method is shown in figure 7.5.

Parallel Jacobi methods require two forms of communication. The columns and/or rows of the matrix must be exchanged in order to compute the rotations and the rotations must be broadcast. The basic communication for one-sided Jacobi is shown in figure 7.7 while the communication pattern for two-sided Jacobi is given in figure 7.8.

7.3.4 Computation costs

The computation and communication cost for the Jacobi method which I recommend for non-vector distributed memory computers with many nodes, a one-sided blocked Jacobi on a 2 dimensional ($p_r \times p_c$) processor grid, is shown in table 7.3. Definitions for all symbols used here can be found in Appendix A.

[7] They also avoid applying rotations from both sides, but this advantage is negated by the fact that they

Figure 7.3: Matlab code for two-sided cyclic Jacobi

```
function [Q,D] = jac2(A)
%
%  Classical two-sided threshold Jacobi
%

thresh = 1e-15;
maxiter = 25;
n = size(A,2)

iter = 0
mods = 1
Q = eye(n);

while (iter < maxiter & mods > 0 )
  mods = 0;
  for I = 1:n
    for J = 1:I-1

      blkA = A([J,I],[J,I]) ;
      if ( norm(blkA-diag(diag(blkA))) > ( norm(blkA)*thresh))
        mods = mods + 1;

        [R,D] = parteig(A([J,I],[J,I]));
        A([J,I],:) = R' * A([J,I],:) ;
        A(:,[J,I]) = A(:,[J,I]) * R ;
        Q(:,[J,I]) = Q(:,[J,I]) * R ;

      end % if
    end % for J
  end % for I
  iter = iter + 1
end % while

D = diag(diag(A)) ;
```

Figure 7.4: Matlab code for two-sided blocked Jacobi

```
function [Q,D] = bjac2( A )
%
%  Two sided blocked threshold Jacobi
%

maxiter = 25 ;
thresh = 1e-15;
nb = 1;
n = size(A,2)

iter = 0;
mods = 1;
Q = eye(n);

while (iter < maxiter & mods > 0 )
  A = ( A + A' ) / 2;             % restore symmetry
  mods = 0;
  for i = 1:nb:n
    maxi = min(i+nb-1,n);
    I = i:maxi;
    for j = 1:nb:I-1
      maxj = min(j+nb-1,n);
      J = j:maxj;

      blkA = A([J,I],[J,I]) ;
      if ( norm(blkA-diag(diag(blkA))) > ( norm(blkA)*sqrt(nb)*thresh))
        mods = mods + 1 ;

        [R,D] = parteig(A([J,I],[J,I])) ;
        A([J,I],:) = R' * A([J,I],:) ;
        A(:,[J,I]) = A(:,[J,I]) * R ;
        Q(:,[J,I]) = Q(:,[J,I]) * R ;

      end % if
    end % for j
  end % for i
  iter = iter + 1
end % while

D = diag(diag(A)) ;
```

Figure 7.5: Matlab code for one-sided blocked Jacobi

```
function [ Q, D ] = bjac1( A )
%
%  One sided blocked Jacobi
%

thresh = 1e-15 ;
nb = 2 ;
maxiter = 25;
n = size(A,2)

B = A;
iter = 0 ;
mods = 1 ;

while (iter < maxiter & mods > 0)
  mods = 0 ;
  for i = 1:nb:n
    maxi = min(i+nb-1,n);
    I = i:maxi;
    for j = 1:nb:I-1
      maxj = min(j+nb-1,n);
      J = j:maxj;

      blkA = A(:,[J,I])' * A(:,[J,I]) ;
      if (norm(blkA-diag(diag(blkA))) > norm(blkA)*sqrt(nb)*thresh)
        mods = mods + 1 ;

        [R,D] = parteig(blkA)  ;
        A(:,[J,I]) = A(:,[J,I]) * R ;

      end % if
    end % for j
  end % for i
  iter = iter + 1
end % while

D = A' * A;
Q = A * diag(1./sqrt(diag(D))) ;

D = Q' * B * Q ;
D = diag(diag(D)) ;
```

Figure 7.6: Matlab code for an inefficient partial eigendecomposition routine

```
%
%  parteig - eigendecomposition with eigenvalues sorted
%
function [ Q, D ] = parteig( A )

[QQ,DD ] = eig(A) ;

[tmp,Index] = sort(- diag(DD));

D = DD(Index,Index) ;
Q = QQ(:,Index) ;
```

Table 7.3: Performance model for my recommended Jacobi method

Task	Cost per parallel pairing	Cost per sweep i.e. $(n/\mathsf{nb}^2)/(2p_c)$ parallel pairings	Cost for recommended data layout ($\mathsf{nb}{=}n/(2p_c)$ $p_c{=}16p_r{=}4\sqrt{p}$)
Move column for this pairing[a]	$2\frac{2\mathsf{nb}p_c}{n}\times$ $(\alpha+\frac{n\mathsf{nb}}{p_r}\beta)$	$\frac{n}{\mathsf{nb}}\alpha+\frac{n^2}{p_r}\beta$	$8\sqrt{p}\alpha+\frac{4n^2}{\sqrt{p}}\beta$
$\mathsf{diag}{=}A([I,J],:)'\times A(I,J],:)$ [b]	$\delta_3+\frac{2n\mathsf{nb}^2}{p_r}\gamma_3$	$(\frac{n}{\mathsf{nb}})^2\frac{1}{2p_c}\delta_3+\frac{n^3}{p}\gamma_3$	$8\sqrt{p}\delta_3+\frac{n^3}{p}\gamma_3$
Sum **diag** within each processor column	$\lg(p_r)\frac{2\mathsf{nb}p_c}{n}\alpha$ $+\lg(p_r)\mathsf{nb}^2\beta$	$(\frac{n}{\mathsf{nb}})\lg(p_r)\alpha$ $+\frac{n^2}{2p_c}\lg(p_r)\beta$	$4\sqrt{p}(\lg(p)-4)\alpha$ $+n^2/(\frac{16}{\sqrt{p}}(\lg(p)-4)\beta$
$[Q,D]=parteig(\mathsf{diag})$ [c]	$2\mathsf{nb}^2(2\gamma_\div+\gamma_\surd)$ $+6(2\mathsf{nb})^3\gamma_1$	$2\frac{n^2}{p_c}\gamma_\div+\frac{n^2}{p_c}\gamma_\surd$ $+\frac{24n^2\mathsf{nb}}{p_c}\gamma_1$	$\frac{1}{2}\frac{n^2}{\sqrt{p}}\gamma_\div+\frac{1}{4}\frac{n^2}{\sqrt{p}}\gamma_\surd$ $+\frac{3}{4}\frac{n^3}{p}\gamma_1$ (see note[d])
Broadcast Q within each processor column	$\lg(p_r)\frac{2\mathsf{nb}p_c}{n}\alpha$ $+\lg(p_r)\mathsf{nb}^2\beta$	$(\frac{n}{\mathsf{nb}})\lg(p_r)\alpha$ $+\frac{n^2}{2p_c}\lg(p_r)\beta$	$4\sqrt{p}(\lg(p)-4)\alpha$ $+\frac{n^2}{16\sqrt{p}}(\lg(p)-4)\beta$
$A=QA$	$\delta_3+2\frac{n}{p_r}(2\mathsf{nb})^2\gamma_3$	$(\frac{n}{\mathsf{nb}})^2\frac{1}{2p_c}\delta_3+4\frac{n^3}{p}\gamma_3$	$8\sqrt{p}\delta_3+4\frac{n^3}{p}\gamma_3$
Total		$\frac{n}{\mathsf{nb}}\alpha+2(\frac{n}{\mathsf{nb}})\lg(p_r)\alpha+$ $\frac{n^2}{p_r}\beta+\frac{n^2}{p_c}\lg(p_r)\beta$ $+2\frac{n^2}{p_c}\gamma_\div+\frac{n^2}{p_c}\gamma_\surd$ $+24\frac{n^2\mathsf{nb}}{p_c}\gamma_1$ $+(\frac{n}{\mathsf{nb}})^2\frac{1}{p_c}\delta_3+5\frac{n^3}{p}\gamma_3$	$8\sqrt{p}(\lg(p)-3)\alpha+\frac{7}{2}\frac{n^2}{\sqrt{p}}\beta+$ $\frac{n^2}{8\sqrt{p}}\lg(p)\beta+\frac{1}{2}\frac{n^2}{\sqrt{p}}\gamma_\div$ $+\frac{1}{4}\frac{n^2}{\sqrt{p}}\gamma_\surd$ $+\frac{3}{4}\frac{n^3}{p}\gamma_1$ $8\sqrt{p}\delta_3+5\frac{n^3}{p}\gamma_3$

[a] My models assume that sends and receives do not overlap, hence the factor of 2. The factor of $(2\mathsf{nb}p_c/n)$ represents the number of parallel pairings that can be performed on the data local to one processor column.

[b] Only $A(I,:)'\times A(J,:)$ need be computed. See section 7.3.8

[c] Partial eigendecomposition of the $(2\mathsf{nb})\times(2\mathsf{nb})$ matrix performed with one pass of an unblocked two-sided Jacobi method exploiting symmetry, see column labeled "exploiting symmetry" in table 7.6

[d] $(\frac{24n^2\mathsf{nb}}{p_c})\times((\frac{n}{\mathsf{nb}})^2\frac{1}{2p_c})=24\frac{n^3}{2p_c}=24/36\frac{n^3}{p}=3/4\frac{n^3}{p}$

Figure 7.7: Pseudo code for one-sided parallel Jacobi with a 2D data layout with communication highlighted

Until convergence do:

  Foreach paring do:

    Move column data (A) to adjacent columns of processors

    Compute $A^T A$ locally (i.e. blkA = A(:,[I,J])' * A(:,[I,J]))

    Combine $A^T A$ within each column of processors

    Partial eigendecomposition of diagonal block (i.e. $[R, D] = eig(A^T A)$)

    Broadcast R within each row of processors

    Compute A R locally

  End Foreach

End Until

Table 7.4 shows the estimated execution time for one sweep of my recommended Jacobi on a matrix of size 1000 by 1000 on a 64-node PARAGON. As this model has not been validated, these estimates must be viewed with caution. Actual performance will be different, but the model gives some idea of how important the various aspects may be. This model is given in matlab form in section B.2.1. Table 7.4 suggests that Jacobi is indeed efficient ($1.68/2.69 = 62\%$) even on such small problems. It also suggests that the optimal data layout may be even taller and thinner than my recommended data layout: $p_c = 32; p_r = 2$. A taller and thinner layout (specifically $p_c = 64; p_r = 1$) would double the cost of message transmission between columns but would decrease the cost of the partial eigensolver. The cost of the divides and square roots in the partial eigensolver would decrease by a factor of 64/32 because all 64 processors would participate in the partial eigensolver. And the cost of accumulating the rotations within the partial eigensolver would decrease by $2 \times 2 = 4$. The first factor of 2 stems the fact that all processors would share in the work, while the second factor of 2 stems from the fact that the block size would be smaller by a factor of 2 and the cost of accumulating rotations grows as $O(n^2 \mathsf{nb})$.

Table 7.5 gives computation cost models for 6 one-sided Jacobi variants. These models are not complete (they overlook many overhead and load imbalance costs), nor have they been validated. This table is designed mainly to put the various variants in perspective and not

must perform dot products to form the square sub matrices to be diagonalized.

Table 7.4: Estimated execution time per sweep for my recommended Jacobi on the PARAGON on n=1000, p=64

Task	Performance Model	Operation cost[a]	Estimated time (seconds)
Message latency	$8\sqrt{p}(\lg(p)-3)\alpha$	$\alpha = 65.9e-6$	0.01
Message transmission between columns	$\frac{7}{2}\frac{n^2}{\sqrt{p}}\beta$	$\beta = .146e-6$	0.06
Message transmission within columns	$\frac{1}{8}\frac{n^2}{\sqrt{p}}\lg(p)\beta$	$\beta = .146e-6$	0.01
Computing rotations	$\frac{1}{2}\frac{n^2}{\sqrt{p}}\gamma_{\div}$	$\gamma_{\div} = 3.85e-6$	0.24
Computing rotations	$\frac{1}{4}\frac{n^2}{\sqrt{p}}\gamma_{\surd}$	$\gamma_{\surd} = 7.7e-6$	0.24
Accumulating rotations in partial eigensolver	$\frac{3}{4}\frac{n^3}{p}\gamma_1$	$\gamma_1 = .074e-6$	0.43
Software overhead	$8\sqrt{p}\delta_3$	$\delta_3 = 103e-6$	0.01
$A = QA$	$5\frac{n^3}{p}\gamma_3$	$\gamma_3 = .0215e-6$	1.68
Total (per sweep)			2.68

[a] See 6.1

Figure 7.8: Pseudo code for two-sided parallel Jacobi with a 2D data layout, as described by Schrieber[150], with communication highlighted

Until convergence do:

  Foreach paring do:

    Move row and column data (A) to diagonally adjacent processors

    Compute partial eigendecomposition of diagonal block

    Broadcast R within each row of processors

    Broadcast R' within each column of processors

    Compute R A R' locally

    Compute Q R locally

  End Foreach

End Until

to establish which is best. Communication costs are considered in section 7.3.5

I have attempted to list the variants that have been implemented as well as the most promising suggestions. For each variant I have, where appropriate, followed my recommendations for implementing a Jacobi code made in section 7.3.16.

Table 7.6 gives performance models for 5 commonly mentioned two-sided Jacobi variants. Like the performance models for one-sided Jacobi variants, these models are incomplete and have not been validated.

7.3.5 Communication costs

Table 7.7 summarizes the communication costs for parallel Jacobi methods. I assume that the communication block size is chosen to be as large as possible.

A performance model for Jacobi could be created by selecting the appropriate computation costs from table 7.5 or table 7.6 and the appropriate communication cost from table 7.7. Not all load imbalance and overhead costs are covered in either of these tables, and the models have not been validated.

Table 7.5: Performance models (flop counts) for one-sided Jacobi variants. Entries which differ from the previous column are shaded.

	Unblocked				Blocked[a]	
	Littlefield Maschhoff[b]	exploit symmetry[c]	store diagonals[d]	fast givens[e]	exploit symmetry[f]	store diagonals[g]
A^TA	$\frac{3}{2}n^2\delta_1{+}3n^3\gamma_1$	$\frac{3}{2}n^2\delta_1{+}3n^3\gamma_1$	$\frac{1}{2}n^2\delta_1{+}n^3\gamma_1$	$\frac{1}{2}n^2\delta_1{+}n^3\gamma_1$	$2p_c^2\delta_3{+}2n^3\gamma_3$	$2p_c^2\delta_3{+}n^3\gamma_3$
$[Q,D]=$ $parteig(A^TA)$ One sweep[h]	$\frac{1}{2}n^2(\gamma_\div{+}\gamma_\surd)$	$\frac{1}{2}n^2(\gamma_\div{+}\gamma_\surd)$	$\frac{1}{2}n^2(\gamma_\div{+}\gamma_\surd)$	$\frac{1}{2}n^2(\gamma_\div{+}\gamma_\surd)$	$8n^2\delta_1{+}$ $n^2(\gamma_\div{+}\gamma_\surd)$ $+24n^2\mathsf{nb}\gamma_1$	$8n^2\delta_1{+}$ $n^2(\gamma_\div{+}\gamma_\surd)$ $+24n^2\mathsf{nb}\gamma_1$
$Q\times L$	$2n^2\delta_1{+}3n^3\gamma_1$	$2n^2\delta_1{+}3n^3\gamma_1$	$2n^2\delta_1{+}3n^3\gamma_1$	$n^2\delta_1{+}2n^3\gamma_1$	$2p_c^2\delta_3{+}4n^3\gamma_3$	$2p_c^2\delta_3{+}4n^3\gamma_3$
$V\times Q$	$2n^2\delta_1{+}3n^3\gamma_1$	0	0	0	0	0
Total (per sweep)	$\frac{7}{2}n^2\delta_1{+}$ $\frac{1}{2}n^2(\gamma_\div{+}\gamma_\surd)$ $+9n^3\gamma_1$	$\frac{7}{2}n^2\delta_1{+}$ $\frac{1}{2}n^2(\gamma_\div{+}\gamma_\surd)$ $+6n^3\gamma_1$	$\frac{5}{2}n^2\delta_1{+}$ $\frac{1}{2}n^2(\gamma_\div{+}\gamma_\surd)$ $+4n^3\gamma_1$	$\frac{3}{2}n^2\delta_1{+}$ $\frac{1}{2}n^2(\gamma_\div{+}\gamma_\surd)$ $+3n^3\gamma_1$	$4p_c^2\delta_3{+}$ $8n^2\delta_1{+}$ $n^2(\gamma_\div{+}\gamma_\surd)$ $+24n^2\mathsf{nb}\gamma_1$ $+6n^3\gamma_3$	$4p_c^2\delta_3{+}$ $8n^2\delta_1{+}$ $n^2(\gamma_\div{+}\gamma_\surd)$ $+24n^2\mathsf{nb}\gamma_1$ $+5n^3\gamma_3$
Assume:[i] $\mathsf{nb}{=}\frac{n}{2p_c}$ $p_c{=}16p_r$	$\frac{7}{8}\frac{n^2}{\sqrt{p}}\delta_1{+}$ $\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_\div{+}$ $\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_\surd{+}$ $9\frac{n^3}{p}\gamma_1$	$\frac{7}{8}\frac{n^2}{\sqrt{p}}\delta_1{+}$ $\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_\div{+}$ $\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_\surd{+}$ $6\frac{n^3}{p}\gamma_1$	$\frac{5}{8}\frac{n^2}{\sqrt{p}}\delta_1{+}$ $\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_\div{+}$ $\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_\surd{+}$ $4\frac{n^3}{p}\gamma_1$	$\frac{3}{8}\frac{n^2}{\sqrt{p}}\delta_1{+}$ $\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_\div{+}$ $\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_\surd{+}$ $3\frac{n^3}{p}\gamma_1$	$64p\delta_3{+}$ $2\frac{n^2}{\sqrt{p}}\delta_1{+}$ $\frac{1}{4}\frac{n^2}{\sqrt{p}}\gamma_\div{+}$ $\frac{1}{4}\frac{n^2}{\sqrt{p}}\gamma_\surd{+}$ $\frac{3}{8}\frac{n^3}{p}\gamma_1{+}$ $6\frac{n^3}{p}\gamma_3$	$64p\delta_3{+}$ $2\frac{n^2}{\sqrt{p}}\delta_1{+}$ $\frac{1}{4}\frac{n^2}{\sqrt{p}}\gamma_\div{+}$ $\frac{1}{4}\frac{n^2}{\sqrt{p}}\gamma_\surd{+}$ $\frac{3}{8}\frac{n^3}{p}\gamma_1{+}$ $5\frac{n^3}{p}\gamma_3$

[a] For parallel codes we assume that the blocksize is chosen to be as large as possible i.e. $\mathsf{nb} = n/(2p_c)$ where p_c is the numer of processor columns. For a serial code $p_c = n/(2\times\mathsf{nb})$ can be arbitrarily chosen.

[b] This is the one-sided method used by Littlefield and Maschoff[125].

[c] This is the method shown in figure 7.5

[d] This is the method used by Arbenz and Oettli[10]

[e] Using fast givens is often mentioned, but rarely implemented. Perhaps the benefit is not as good as this model would suggest.

[f] This is the method shown in figure 7.4

[g] This is the method used by Arbenz and Slapnicar[9]

[h] One sweep of Jacobi on an matrix of size $2\mathsf{nb}$ by $2\mathsf{nb}$

[i] I also assume that only one processor in each processor column is involved in each partial eigendecomposition.

Table 7.6: Performance models (flop counts) for two-sided Jacobi variants

	Unblocked			Blocked[a]	
	Ignore symmetry[b]	Exploit symmetry	fast givens[c]	Ignore symmetry[d]	Exploit symmetry
$parteig(A([I,J],[I,J])$ (one sweep[e])	$\frac{1}{2}n^2(\gamma_{\div}+\gamma_{\surd})$	$\frac{1}{2}n^2(\gamma_{\div}+\gamma_{\surd})$	$\frac{1}{2}n^2(\gamma_{\div}+\gamma_{\surd})$	$8n^2\delta_1+$ $n^2(\gamma_{\div}+\gamma_{\surd})$ $+24n^2\mathsf{nb}\gamma_1$	$8n^2\delta_1+$ $n^2(\gamma_{\div}+\gamma_{\surd})$ $+24n^2\mathsf{nb}\gamma_1$
QAQ^T Rotate from both sides	$4n^2\delta_1+6n^3\gamma_1$	$2n^2\delta_1+$ $3n^3\gamma_1$	$n^2\delta_1+2n^3\gamma_1$	$4p_c^2\delta_3+8n^3\gamma_3$	$4p_c^2\delta_3+4n^3\gamma_3$
QZ Update eigenvectors	$2n^2\delta_1+3n^3\gamma_1$	$2n^2\delta_1+3n^3\gamma_1$	$n^2\delta_1+3n^3\gamma_1$	$2p_c^2\delta_3+4n^3\gamma_3$	$2p_c^2\delta_3+4n^3\gamma_3$
Total (per sweep)	$\frac{1}{2}n^2(\gamma_{\div}+\gamma_{\surd})$ $+6n^2\delta_1$ $+9n^3\gamma_1$	$\frac{1}{2}n^2(\gamma_{\div}+\gamma_{\surd})$ $+4n^2\delta_1$ $+6n^3\gamma_1$	$\frac{1}{2}n^2(\gamma_{\div}+\gamma_{\surd})$ $+2n^2\delta_1$ $+4n^3\gamma_1$	$8n^2\delta_1+$ $n^2(\gamma_{\div}+\gamma_{\surd})$ $+24n^2\mathsf{nb}\gamma_1+$ $6n^2\delta_1+12n^3\gamma_1$	$8n^2\delta_1+$ $n^2(\gamma_{\div}+\gamma_{\surd})$ $+24n^2\mathsf{nb}\gamma_1+$ $6n^2\delta_1+8n^3\gamma_1$
Assume:[f] $\mathsf{nb}=\frac{n}{2p_c}$ $p_c=16p_r$	$\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_{\div}+$ $\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_{\surd}+$ $\frac{3}{2}\frac{n^2}{\sqrt{p}}\delta_1+$ $9\frac{n^3}{p}\gamma_1$	$\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_{\div}+$ $\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_{\surd}+$ $\frac{n^2}{\sqrt{p}}\delta_1+$ $6\frac{n^3}{p}\gamma_1$	$\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_{\div}+$ $\frac{1}{8}\frac{n^2}{\sqrt{p}}\gamma_{\surd}+$ $\frac{1}{2}\frac{n^2}{\sqrt{p}}\delta_1+$ $4\frac{n^3}{\sqrt{p}}\gamma_1$	$2\frac{n^2}{\sqrt{p}}\delta_1+$ $\frac{n^2}{\sqrt{p}}\gamma_{\div}+$ $\frac{1}{4}\frac{n^2}{\sqrt{p}}\gamma_{\surd}+$ $\frac{3}{8}\frac{n^3}{p}\gamma_1+$ $\frac{3}{2}\frac{n^2}{\sqrt{p}}\delta_1+$ $12\frac{n^3}{p}\gamma_1$	$2\frac{n^2}{\sqrt{p}}\delta_1+$ $\frac{1}{4}\frac{n^2}{\sqrt{p}}\gamma_{\div}+$ $\frac{1}{4}\frac{n^2}{\sqrt{p}}\gamma_{\surd}+$ $\frac{3}{8}\frac{n^2}{p}\gamma_1+$ $\frac{1}{2}\frac{n^2}{\sqrt{p}}\delta_1+$ $8\frac{n^3}{p}\gamma_1$

[a] For parallel codes we assume that the blocksize is chosen to be as large as possible i.e. $\mathsf{nb} = n/(2p_c)$ where p_c is the number of processor columns. For a serial code $p_c = n/(2 \times \mathsf{nb})$ can be arbitrarily chosen.

[b] This is the method used by Pourandi and Tourancheau[142], by Schreiber[150] and the method described in figure 7.3.

[c] Using fast givens is often mentioned, but rarely implemented. Perhaps the benefit is not as good as this model would suggest.

[d] This is the method shown in figure 7.4

[e] One sweep of Jacobi on an matrix of size 2nb by 2nb

[f] I also assume that only one processor in each processor column is involved in each partial eigendecomposition.

Table 7.7: Communication cost for Jacobi methods (per sweep)

	One-sided		Two-sided		
	1-D data layout[a]	2-D data layout[b]	1-D data layout[c]	2-D data layout[d]	2-D data layout[e]
exchange column vectors	$4p\alpha + 2n^2\beta$	$4p_c\alpha + 2\frac{n^2}{p_r}\beta$	$4p\alpha + 2n^2\beta$	$6p_c \log(\sqrt{p})\alpha + \frac{3}{2}\frac{n^2 \log(\sqrt{p})}{p_r}\beta$	$12\sqrt{p}\alpha + 3\frac{n^2}{\sqrt{p}}\beta$
Reduce $A^T A$	0	$2p_c \lg(p_r)\alpha + \frac{2n^2}{p_c}\lg(p_r)\beta$	0	0	0
Broadcast rotations[f]	0	$2p_c \lg(p_r)\alpha + \frac{1}{2}n^2 \lg(p_r)\beta$	$2p \lg(p)\alpha + \frac{n^2}{p}\lg(p)\beta$	$4p_c \log(p_r)\alpha + 4p_c \log(p_c)\alpha + \frac{1}{2}\frac{n^2}{p_r}\log(p_c)\beta + \frac{1}{2}\frac{n^2}{p_c}\log(p_r)\beta$	$8\sqrt{p}\alpha + 2\frac{n^2}{\sqrt{p}}\beta$

[a]This is the method used by Arbenz and Slapnicar[9]
[b]This is the method used by Littlefield and Maschhoff[125]
[c]This is the method used by Pourzandi and Tourancheau[142]
[d]This is 2D method most likely to be used today
[e]This is method used by Schreiber[150]
[f]On the unblocked methods we assume that communication is blocked even though the computation is not. We also assume that each rotation is sent as a single floating point number. This is natural if you are using fast Givens but requires extra divides and square roots if fast Givens are not used.

7.3.6 Blocking

Classical Jacobi methods annihilate individual entries whereas blocked Jacobi methods use a partial eigendecomposition on blocks. Cyclic Jacobi methods use fewer flops, especially if fast Givens rotations are used. But, almost all of the floating point operations in blocked Jacobi methods are performed in matrix-matrix multiply operations, the most efficient operation.

Both cyclic and blocked Jacobi methods can be blocked for communication. The communication block size need only be an integer multiple of the computation block size. Blocking for communication may be more important than blocking for computation because it reduces the number of messages by a factor equal to the communication block size.

Blocking allows greater possibilities for the partial eigendecomposition. A better partial eigendecomposition will lead to faster convergence. For example, performing two Jacobi sweeps in the partial eigendecomposition would result in fewer sweeps through the entire matrix. However initial experiments indicate that on random matrices the best that one can hope for is a reduction of $\lg(\mathsf{nb})$ in the number of full sweeps even if one uses a complete eigendecomposition as the "partial eigendecomposition".

Using a block size that is smaller than the maximum allowed (i.e. $\mathsf{nb} < n/(2p_c)$) offers various possibilities. It allows communication to be pipelined to some extent. Alternatively it allows more than p_c processors to be involved in computing the partial eigendecompositions.

The per sweep cost of the partial eigensolutions grows as the square of the block size because larger block sizes mean that fewer processors are involved in the partial eigendecomposition[8].

I recommend keeping the code simple by keeping the communication and computation block size equal and setting $\mathsf{nb} = n/(2p_c)$ so that each parallel pairing involves one partial eigendecomposition per processor column. Using a rectangular process grid such that $(16p_r \leq p_c \leq 32p_r)$ requires a lower nb and hence allows the code to keep communication and computation block size equal while holding the cost of the partial eigendecomposition to $\frac{3}{8}$ to $\frac{3}{4}\frac{n^3}{p}\gamma_1$. On most machines this will be no more than half the $\frac{5n^3}{p}\gamma_3$ cost, in part because the partial eigendecomposition will fit in the highest level data cache.

A larger computational block size increases the cost of partial eigendecomposition and decreases the cost of the BLAS3 operations. Larger communication block size decreases message latency cost but leaves less opportunity for overlapping communication with computation. A larger ratio of p_c to p_r increases message latency but reduces the partial eigendecomposition cost[9]. See section 7.3.9 for details on the partial eigendecomposition cost.

7.3.7 Symmetry

Exploiting symmetry in two-sided Jacobi methods is important because it reduces the number of flops per sweep from $12n^3$ to $8n^3$. However, exploiting symmetry while maintaining load balance is difficult. If in a blocked Jacobi method, the block size were set to the largest value possible, i.e. $\frac{n}{2p_c}$, and a standard rectangular grid of processors were used, half of the processors (either those above or below the diagonal) would be idle all the time. Using a smaller block size would allow better load balance but gives up some of the benefits of blocking. Alternatives, such as using a different processor layout for the eigenvector update, are feasible, but their complexity make them unattractive.

[8] This does not hold for $\mathsf{nb} < n/(2p)$.

[9] Assuming only one processor per processor column is involved in computing partial eigendecompositions.

In one-sided Jacobi methods, A^TA is symmetric and only half of it need be computed. In fact, only a quarter of it must be computed as shown in the following section.

7.3.8 Storing diagonal blocks in one-sided Jacobi

One-sided Jacobi methods must compute diagonal blocks of A^TA. This is shown in the matlab code given in figure 7.5 as: $blkA = A(:,[J,I])' * A(:,[J,I])$. This is inefficient because not only does it compute both halves of a symmetric (or hermitian) matrix, but $A[:,I]' * A[:,I]$ and $A[:,J]' * A[:,J]$ are already known. They are the diagonal blocks returned by `parteig` on the most recent previous pairing which involved I and J respectively. Storing these blocks for future use avoids the need to recompute them, although they may need to be refreshed from time to time for accuracy reasons.

7.3.9 Partial Eigensolver

My performance models suggest that execution time is likely to be minimized when the partial eigendecomposition consists of either one or two sweeps of Jacobi. The per sweep cost of the partial eigensolver grows as $O(\frac{n^2 \mathsf{nb}}{\sqrt{p}})$. In my recommended Jacobi method, the partial eigensolver consists of one sweep of Jacobi and based on the data layout which I recommend, and costs $\frac{3}{8}\frac{n^3}{p}\gamma_1 + O(\frac{n^2}{\sqrt{p}})$ or roughly 10% to 30% of the total cost of the sweep. Preliminary experiments indicate that with a block size of 32, using a full eigendecomposition instead of a partial eigendecomposition may reduce the number of sweeps by as much as 20%. Assuming that a full eigendecomposition of a 32 by 32 matrix costs 6 times what a single sweep of Jacobi would cost, this analysis suggests that the added cost of a full eigendecomposition will not reduce the number of sweeps sufficiently to result in a net decrease in execution time, especially if `DGEMM` performs efficiently on a smaller block size[10]. On the other hand, since most of the advantage of a full eigendecomposition will come from the second sweep, using two sweeps of Jacobi in the partial eigensolver may result in a net decrease in execution time. This analysis depends on a great many assumptions and should be taken as a guide, not a prediction. Schreiber[150] reached a similar conclusion.

In a non-blocked code, the "partial eigendecomposition" should consist of a rotation, i.e. a full eigendecomposition. In a non-blocked code, the cost of the partial eigensolver,

[10] A smaller block size reduces the cost of the partial eigensolver.

though still $O(n^2\mathsf{nb})$, is lower because $\mathsf{nb} = 1$ and for a 2 by 2 matrix, a single sweep of Jacobi is a full eigendecomposition. Except for very small n, say $n < 100$, partial eigendecompositions, such as those suggested by Götze[85], are not likely to result in lower total execution time.

In a blocked eigensolver, one must compute a partial eigendecomposition for each pairing. Most commonly, a single sweep of two-sided Jacobi is used as the partial eigendecomposition. Since the elements in $A[I,I]' \times A[I,I]$ and $A[J,J]' \times A[J,I]$ are involved in more pairings than the elements in $A[I,J]' \times A[I,J]$ they need not be annihilated in every pairing.

The number of partial eigenproblems that can be performed simultaneously is $\frac{n}{2\mathsf{nb}}$. If this is less than p, either the partial eigenproblems must themselves be performed in parallel or some processors will be idle. Unless nb is quite large, say $\mathsf{nb} \geq 64$, it is likely to be faster to compute them each on a single processor, especially since the partial eigendecomposition is a two-sided, not one-sided, sweep.

If $n/(2\mathsf{nb}) = p_c$, it is natural to assign one processor within each processor column to perform the partial eigendecomposition. If $n/(2\mathsf{nb}) > p_c$, each parallel pairing will have more partial eigenproblems than processor columns, hence the code could assign different partial eigenproblems to different processors within each processor column. The other alternative is to increase p_c (decreasing p_r). Hence, assigning different partial eigenproblems to different processors within a column only makes sense if bandwidth cost makes increasing p_c unattractive. On the other hand, the only disadvantage to assigning different partial eigenproblems to different processors with a column (as opposed to increasing p_c) is increased code complexity.

If the cost of divisions and square roots ($\frac{1}{4}n^2\frac{2\gamma_{\div}+\gamma_{\surd}}{p_c}$) is significant, one should consider inexact rotations in the partial eigensolver. Götze points out that one need not perform exact rotations and suggests a number of approximate rotations which avoid divides and square roots[85]. It would be counterproductive to use inexact rotations (saving $O(n^2)$ flops at the expense of increasing the number of sweeps and the accompanying $O(n^3)$ flops) in a parallel cyclic Jacobi method. Likewise I would be hesitant to use inexact rotations in the partial eigensolver unless doing so makes it feasible to perform two sweeps in the partial eigensolver. However, it is entirely possible that more sweeps with inexact rotations might be better than fewer sweeps using exact rotations in the partial eigensolver.

Using a classical threshold scheme in the partial eigensolver is likely to save little time, but using thresholds to perform more important rotations might improve performance. A classical threshold scheme is not attractive because the processors performing fewer rotations would simply sit idle. However having each processor compute the same number of rotations, while using thresholds to skip some rotations might allow the rotations performed to be more productive.

7.3.10 Threshold

For serial cyclic codes, thresholds can significantly reduce the total number of floating point operations performed, especially on spectrally diagonally dominant matrices. Since Jacobi methods are most likely to be attractive on spectrally diagonally dominant codes, thresholds cannot be rejected as unimportant. However in a blocked parallel program, entire blocks can only be skipped if the whole block requires no rotations. As an example, consider a blocked parallel Jacobi eigensolution of a 1024 by 1024 matrix on a 1024 node computer using a block size of 16. This would involve 63 (or 64) steps each of which would consist of 32 pairings performed in parallel. Each pairing involves a partial eigendecomposition of a 2×16 by 2×16 matrix. If any of the off-diagonal elements in any of the 32 pairings requires annihilation, no savings is achieved in that step. Hence, in the worst case, if just 63 of the 499,001 off-diagonal elements (one per step) require annihilation, the threshold algorithm realizes no benefit.

Corbato[47] devised a method for implementing a classical Jacobi method in $O(n^3)$ time. His method involves keeping track of the largest off-diagonal element in each column. The cost of maintaining this data structure would more than double the cost of each rotation and may not lead to reduced execution time even in serial codes. However, Beresford Parlett[137] pointed out to me that one need not keep track of the true largest element and that each rotation must maintain the sum of the squares of the elements, hence allowing the list of "largest" off-diagonal elements to be out-of-date would seriously undermine the advantage and would significantly reduce the overhead. This deserves further study.

Untested Threshold methods

One could design a code that used variable block sizes and/or switched from a one-sided-non-threshold Jacobi to a two-sided-threshold Jacobi. A code could even scan the matrix, identify the elements that need to be eliminated and select pairings and block sizes that would eliminate those elements as efficiently as possible. In our worst case example given in the preceding paragraph, it might be that those 63 off diagonal elements could be annihilated in just two parallel steps each requiring only a two element rotation.

Scanning all off-diagonal elements and choosing the largest n non-interfering elements might be an attractive compromise between the classical Jacobi method which examines all off diagonal elements and annihilates the largest and the cyclic Jacobi method which annihilates all elements without regard to size. If software overhead could be kept modest, such a method might on small spectrally diagonally dominant matrices. Precisely the matrices that are best suited to Jacobi methods.

Jacobi methods that attempt to annihilate larger elements, i.e. threshold methods, work best on two-sided Jacobi methods. This is unfortunate because it appears that one-sided Jacobi is otherwise preferred.

As mentioned in section 7.3.9, thresholds might be useful in the partial eigendecomposition.

7.3.11 Pairing

The order in which the off-diagonal elements are annihilated is referred to as the pairing strategy. Eliminating off-diagonal element $A_{i,j}$ in a two-sided Jacobi requires that rows i and j of A and columns i and j of A be rotated. Hence, rows i and j of A must be distributed similarly, i.e. $A_{i,k}$ and $A_{j,k}$ must both reside on the same processor. Likewise, columns i and j of A must be distributed similarly. Orthogonalizing vectors i and j in a one-sided Jacobi also requires that the two vectors be distributed similarly. In order to annihilate multiple off-diagonal elements simultaneously, they must reside on different sets of processors.

The pairing strategy affects execution time through communications cost, number of pairings per sweep and number of sweeps required for convergence. Different pairing strategies require different communication patterns and hence different communication costs. Some pairings strategies require slightly more pairings than others. Mantharam and Eberlein argue that some pairings lead to faster convergence than others[72].

In this section, we illustrate two pairing strategies, showing how each would pair 8 elements in 4 sets at a time. The elements might be individual indices (in a non-blocked Jacobi) or blocks of indices. The sets might correspond to individual processors (in a 1D data layout) or columns of processors (in a one-sided Jacobi on a 2D layout) or rows and columns of processors (in a two-sided Jacobi on a 2D layout). Furthermore, several sets might be assigned to the same processor or column of processors.

The classic round robin pairing strategy[84] leaves one element stationary and rotates the other elements. As the following diagram shows, in 7 pairings, each element is paired exactly once with each of the other elements. Elements 3 through 8 follow elements 2 through 7 respectively, while element 2 follows element 8.

$$\begin{array}{|c|c|c|c|} 1 & 2 & 3 & 4 \\ 8 & 7 & 6 & 5 \end{array}$$
$$\begin{array}{|c|c|c|c|} 1 & 3 & 4 & 5 \\ 2 & 8 & 7 & 6 \end{array}$$
$$\begin{array}{|c|c|c|c|} 1 & 4 & 5 & 6 \\ 3 & 2 & 8 & 7 \end{array}$$
$$\begin{array}{|c|c|c|c|} 1 & 5 & 6 & 7 \\ 4 & 3 & 2 & 8 \end{array}$$
$$\begin{array}{|c|c|c|c|} 1 & 6 & 7 & 8 \\ 5 & 4 & 3 & 2 \end{array}$$
$$\begin{array}{|c|c|c|c|} 1 & 7 & 8 & 2 \\ 6 & 5 & 4 & 3 \end{array}$$
$$\begin{array}{|c|c|c|c|} 1 & 8 & 2 & 3 \\ 7 & 6 & 5 & 4 \end{array}$$

A slight variation, called the caterpillar pairing method[72, 73, 150], cuts the communication cost in half at the expense of increasing the number of pairings from $n-1$ to n. The caterpillar method, modified so that communication is always performed in the same direction, is shown below. Only the elements in the top line rotate, and they always rotate to the left. The elements, shown in red, in the bottom line get swapped into the top line one at a time. In this pairing method, it takes 8 pairings in order for each element to be paired with every other element. The swapped elements need not perform any work, but must exchange the blocks assigned to them prior to the next communication step. This pairing strategy requires 16 (in general $2n$) pairings to come back to the original pairing, but the second n pairings duplicate the first n.

$$\begin{array}{|c|c|c|c|} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{array}$$

$$\begin{array}{|c|c|c|c|} 2 & 3 & 4 & 8 \\ 5 & 6 & 7 & 1 \end{array}$$

$$\begin{array}{|c|c|c|c|} 3 & 4 & 8 & 2 \\ 5 & 6 & 7 & 1 \end{array}$$

$$\begin{array}{|c|c|c|c|} 4 & 8 & 7 & 3 \\ 5 & 6 & 2 & 1 \end{array}$$

$$\begin{array}{|c|c|c|c|} 8 & 7 & 3 & 4 \\ 5 & 6 & 2 & 1 \end{array}$$

$$\begin{array}{|c|c|c|c|} 7 & 6 & 4 & 8 \\ 5 & 3 & 2 & 1 \end{array}$$

$$\begin{array}{|c|c|c|c|} 6 & 4 & 8 & 7 \\ 5 & 3 & 2 & 1 \end{array}$$

$$\begin{array}{|c|c|c|c|} 5 & 8 & 7 & 6 \\ 4 & 3 & 2 & 1 \end{array}$$

$$\begin{array}{|c|c|c|c|} 8 & 7 & 6 & 5 \\ 4 & 3 & 2 & 1 \end{array}$$

Mantharam and Eberlein[72] suggest that some pairing strategies may lead to convergence in fewer steps than others.

7.3.12 Pre-conditioners

One sided Jacobi methods compute eigenvectors by orthogonalizing a matrix which has the same or related left singular vectors as the original matrix. Some options include:

$[U, D, V] = \mathsf{svd}(A)$; U contains the eigenvectors of A, D is the absolute value of the eigenvalues of A. This method is used by Berry and Sameh[21].

$[U, D, V] = \mathsf{svd}(\mathsf{chol}(A))$; U contains the eigenvectors of A, D is the square root of the eigenvalues of A. This is used by Arbenz and Slapničar[9] and is mathematically equivalent to classical Jacobi.

$[Q, R] = \mathsf{qr}(A); [U, D, V] = \mathsf{svd}(R); Q * U$ contains the eigenvectors of A. D contains the absolute value of the eigenvalues of A

In addition, there are pivoting counterparts to both Cholesky and QR, indeed many flavors of QR with pivoting, which would improve these pre-conditioners. If A is spectrally diagonally dominant, permuting A so that the diagonal elements are non-increasing might provide most of the benefit that Cholesky with pivoting does and at considerably lower cost.

7.3.13 Communication overlap

Overlapping communication and computation is attractive because in theory it reduces the total cost from the sum of the computation and communication costs to their maximum. Arbenz and Slapničar demonstrated that overlapping communication and computation is straightforward in a one-sided Jacobi method with a one-dimensional data layout[10]. But, overlapping communication and computation when using a two-dimensional data layout is not as straightforward. Furthermore, actual experience with communication and computation overlap has been disappointing, see section B.1.6

7.3.14 Recursive Jacobi

The partial eigendecomposition could be a recursive call to a Jacobi eigensolver. A recursive Jacobi could offer all the benefits shown by Toledo on LU[165], notably excellent use of the memory hierarchy. Unfortunately, each level of recursion requires 6 calls, tripling the software overhead. Therefore, the number of subroutine calls, and hence the software overhead, grows at an unacceptably high $O(n^{\lg(6)})$.

Increasing software overhead in order to reduce the number of sweeps will make sense for large matrices but not for small matrices. Since Jacobi is unlikely to be faster than tridiagonal based methods for large matrices, I feel that it is more important to concentrate on making Jacobi fast on smaller matrices. Hence, I do not include recursion as a part of my recommended Jacobi method. Nonetheless, it may be that one step of recursion (tripling the software overhead) and conceivably two steps of recursion (increasing software overhead by a factor of 9) may reduce total execution time, but I would not expect the improvement to be significant.

7.3.15 Accuracy

Demmel and Veselić[58] prove that on scaled diagonally dominant matrices, Jacobi can compute small eigenvalues with high relative accuracy while tridiagonal based methods cannot. Drmač and Veselić[71] show that Jacobi methods can be used to refine an eigensolution, thereby providing high relative accuracy on scaled diagonally dominant matrices at lower total cost than a full Jacobi. Demmel et al.[56] give a comprehensive discussion of the situations in which Jacobi is more accurate than other available algorithms.

7.3.16 Recommendation

If I were asked to write one Jacobi method for all non-vector distributed memory computers, it would be a one-sided blocked Jacobi method. It would use a one-dimensional data layout on computers with fewer than 48 nodes and a two-dimensional data layout on computers with 48 or more nodes. It would use 16-32 times as many processor columns as rows in a two-dimensional data layout[11]. It would use a computational and communication block size equal to[12] $\max(n/(2p_c), 8)$, leaving processors idle if $8 < n/(2p_c)$. It would compute the partial eigendecompositions on just one processor in each processor column. It would avoid recomputing diagonal entries unnecessarily, use a one-directional caterpillar track pairing and one sweep of Jacobi for the partial eigendecomposition. It would use the largest block size possible for both computation and communication.

If I had time to experiment, I would investigate different partial eigendecompositions, pre-conditioners and pairing strategies in that order. Overlapping communication and computation appears to offer greater performance improvements in theory than in practice. I would use thresholds as a part of the stopping criteria, but wouldn't count on them to avoid unnecessary flops. I would check to make sure that my suggested data layout (1D for $p < 48$, $16p_r < p_c < 32p_r$ for $p > 48$ and $\texttt{nb} = \max(n/(2p_c), 8)$) was reasonable on several computers, but unless there was a substantial benefit to tuning the data layout to each machine I would hesitate to do so.

For vector machines I recommend an unblocked code with fast Givens rotations if the cost of `BLAS1` operations is no more than twice that of `BLAS3` operations. If the `BLAS1` operations cost just twice what `BLAS3` operations cost, the flop cost in an unblocked

[11] The ratio $\frac{p_c}{p_r}$ can be made to fall in the 16-32 range for any number of processors except 1 to 15, 32 to 63 and 128 to 144. No more than 2.1% of the processors are left idle following these rules.

[12] Definitions for all symbols used here can be found in Appendix A.

code would be 6/5 that of the blocked code (because unblocked codes using fast Givens require 3/5 as many flops. Savings on other aspects can be expected to make up for this difference on all but the largest matrices. Communication should still be blocked however. One-dimensional data layout can be used for more nodes if a cyclic code is used, perhaps as many as a hundred nodes, since block size is not an issue. As long as $n < 2p$ a one-dimensional data layout is limited only by communication costs.

Combining elements of classical and cyclic Jacobi is an interesting long shot. Classical Jacobi always annihilates the largest off-diagonal element but requires $O(n^4)$ comparisons per sweep[13]. Annihilating the n largest off-diagonal elements each time would roughly match the number of comparisons performed to the number of flops performed. To parallelize this idea, one would have to choose the n largest *non-interfering* elements.

7.4 ISDA

The total execution time for the ISDA[97] for solving the symmetric eigenproblem[14] will be no less than $100n^3$ on typical matrices. The execution time depends largely on how many decouplings are required to make each of the smaller matrices no larger than half the size of the original matrix. It also depends on the cost of each decoupling, but this will not vary that much.

The ISDA achieves high floating point execution rates, but in order to beat tridiagonal methods it must achieve $100/(10/3) = 30$ times higher floating point rates, which it does not. The PRISM implementation of ISDA takes 36 minutes = 2160 seconds to compute the eigendecomposition of a matrix of size 4800 by 4800 on the 100 node SP2 at Argonne[29], `ScaLAPACKs` `PDSYEVX` takes 397 seconds to compute the eigendecomposition of a matrix of size 5000 by 5000 on a 64 node SP2[31]. ISDA should not require as large a granularity, $n/\sqrt{p}$, as `PDSYEVX` because of its heavy reliance on matrix-matrix multiply. However, at present, the PRISM implementation is still at least three times slower than `PDSYEVX` even on small matrices. Solving a matrix of size 800 by 800 on 64 nodes takes 60 seconds using the PRISM ISDA code, whereas `PDSYEVX` can solve a matrix of size 1000 by 1000 on 64 nodes of an SP2 in 16 seconds.

The cost of each decoupling depends upon how close the split happens to come to

[13] Or increased overhead if Corbato's method[47] is used.
[14] See section 2.7.3 for a brief description of the ISDA.

a eigenvalue of the matrix being split. The number of beta function evaluations required for a given decoupling is roughly $-\log(min_{i\in n}(split - \lambda_i))$, where $split$ is the split point selected for this decoupling. The distance between $split$ and the nearest eigenvalue cannot be computed in advance but is likely to fall in the range: $(\log(n)/\log(1.5)+2, \log(n)/\log(1.5)+8$. This is consistent with empirical results. For our purposes we will say that the number of beta function evaluations is: $(\log(1500)/\log(1.5)+2 = 20$. The cost per beta function evaluation is 2 matrix-matrix multiplies at: $2(n')^3/p\gamma_3$ each, where n' is the size of the matrix being decoupled. Hence the cost for the first decoupling is: $2\times 2\times 20n^3/p\gamma_3 = 80n^3/p\gamma_3$.

If each decoupling splits the matrix exactly in half, round i of decouplings involves 2^i decouplings each involving a matrix of size $n/2^i$ at a total cost of: $2^i \times 80(n/2^i)^3 = 80n^3/4^i$. The sum of all rounds would then be: $\sum_{i=0}^{\infty} 80n^3/4^i = 80\times 4/3 = 107n^3$.

The ISDA for symmetric eigendecomposition may require substantially longer on some matrices with a single cluster of eigenvalues containing more than half of the eigenvalues and on matrices with most of the eigenvalues at one end of the spectrum[15]. It is unlikely that the first split point chosen for decoupling will lie in the middle of a cluster. Hence, if the matrix contains one large cluster, that cluster will likely remain completely in one of the two submatrices, making the decoupling less even and hence less successful. Likewise, if most of the eigenvalues are at one end of the spectrum, the submatrix on that end of the spectrum will likely be much larger than the other after the first decoupling. If each decoupling splits off only 20% of the spectrum, the total time will be twice what it would be if each decoupling splits the spectrum exactly in half.

One could check to make sure that a reasonable split point has been chosen by performing an LDL^T decomposition on the shifted matrix, and counting the number of positive or negative values in D. An LDL^T decomposition costs $1/3n^3$ flops or about 0.5% of the flops required to perform the full decoupling.

7.5 Banded ISDA

Banded ISDA is very nearly a tridiagonal based method and hence offers performance that is nearly as good as tridiagonal based methods. PRISM's single processor implementation of banded ISDA is two to three times slower than bisection (`DSTEBZ`)[26].

[15] Fann et al.[75] present a couple examples of real applications that fit this description.

Computing eigenvectors using banded ISDA will not only be more difficult to code, it will require about twice as many flops as inverse iteration. Banded ISDA requires additional bandwidth reductions, each of which requiring up to $2n^3$ additional flops during back transformation[16].

Banded ISDA could make sense if reduction to banded form were twice as fast as reduction to tridiagonal. Although even then one has to question whether it makes sense to use banded ISDA instead of a banded solver.

Banded ISDA should perform a few shifted LDL^T decompositions to make sure that the selected shift will leave at least 1/3 of the matrix in each of the two submatrices.

7.6 FFT

Yau and Lu[174] have implemented an FFT based invariant subspace decomposition method. It, like ISDA, uses efficient matrix-matrix multiply flops, but since it requires $100n^3$ flops the same analysis which shows that ISDA will not be faster applies to it as well. Domas and Tisseur have implemented a parallel version of the Yau and Lu method[60].

[16] The first bandwidth reduction essentially always requires the full $2n^3$ flops during back transformation, though later ones typically require less than that. However, taking advantage of the opportunity to perform fewer flops wither means a complex data structure or that the update matrix Q be formed and then applied, adding another $4/3n^3$ flops.

Chapter 8

Improving the `ScaLAPACK` symmetric eigensolver

8.1 The next `ScaLAPACK` symmetric eigensolver

The next `ScaLAPACK` symmetric eigensolver will be 50% faster than the `ScaLAPACK` symmetric eigensolver in version 1.5 and provide performance that is independent of the user's data layout. Separating internal and external data layout will not only make the code easier to use because the user need not modify their storage scheme, it will also improve performance. The next `ScaLAPACK` symmetric eigensolver will select the fastest of four methods for reduction to tridiagonal form[1], and use Parlett and Dhillon's new tridiagonal eigensolver[139].

Separating internal and external data layout allows execution time to be reduced for three reasons. It allows reduction to tridiagonal form and back transformation to use different data layouts. It allows reduction to tridiagonal form to use a square processor grid, significantly reducing message latency and software overhead. It allows the code to support any input and output data layout without all the layers of software required to support any data layout. Last but not least by concentrating our coding efforts on the simple, but efficient square cyclic data layout, we can implement several reduction to tridiagonal codes and incorporate ideas that would be prohibitively complicated in a code that had to support multiple data layouts.

[1] On machines where timers are not available, a heuristic will be used which may not always pick the fastest.

The rest of this section concentrates on improving execution time in reduction to tridiagonal form. Back transformation is already very efficient and hence leaves less room for improvement. We leave the tridiagonal eigensolver to others[139]. Figure 8.1 gives a top-level description of the next ScaLAPACK symmetric eigensolver.

Figure 8.1: Data redistribution in the next ScaLAPACK symmetric eigensolver

Choose a data layout for reduction to tridiagonal form (see figure 8.2)
Redistribute A to reduction to tridiagonal form data layout
Reduce to tridiagonal form
Replicate diagonal, (D), and sub-diagonal, (E), to all processors
Use Parlett and Dhillon's tridiagonal eigendecomposition scheme
Choose data layout for back transformation
    BCK-$p_r = \lceil \sqrt{p/15} \rceil$; BCK-$p_c = \lfloor p/p_r \rfloor$; BCK-nb $= \lceil n/(k\, p_c) \rceil$
If space is limited, redistribute A back to original data layout
Redistribute eigenvectors, Z, to back transformation data layout
Redistribute A to back transformation data layout
Perform back transformation
Redistribute eigenvectors to user's format

8.2 Reduction to tridiagonal form in the next ScaLAPACK symmetric eigensolver

Figure 8.2 shows how the data layout for reduction to tridiagonal form will be chosen. The data layout and the code used for reduction to tridiagonal form must be chosen in tandem.

Although the new PDSYTRD has three variants, they all share the same pattern of communication and computation shown in figure 8.3.

Message initiations are reduced by using techniques first used in HJS, and several new ones. HJS stores V and W in a row-distributed/column-replicated manner which avoids to need to broadcast them repeatedly. HJS also keeps the number of messages small by combining messages wherever possible.

Our communication pattern has three advantages over HJS: it requires fewer messages, does not risk over/underflow and uses only the BLACS communication primitives[2]. The manner in which we compute the Householder vector requires the same number of message initiations as the HJS, but avoids the risk of over/underflow in the computation of the norm. We use fewer messages than HJS because we update w in a novel manner (see

[2]Whether the right communication primitives were chosen for the BLACS may be debatable, but they are what is available for use within ScaLAPACK.

Figure 8.2: Choosing the data layout for reduction to tridiagonal form

```
If timers (or environmental inquiry routine) are available
    Time select operations
    Determine the best data layout for each of the four reduction to tridiagonal form codes
    Estimate the execution time for each of the four reduction to tridiagonal form codes
    Select the fastest code and the corresponding data layout
else
    if $p/\lfloor\sqrt{p}\rfloor^2 \geq 1.5$ (i.e if p =2,3,6,7,14, 15)
        TRD-$p_r = \lfloor p/7.5 \rfloor + 1$
        TRD-$p_c = p/p_r$
        TRD-nb = 32
        Use old PDSYTRD
    else
        TRD-$p_c = \lfloor\sqrt{p}\rfloor$
        TRD-$p_r$ =TRD-$p_c$
        TRD-nb = 1
        if the compiler is good
            if ($n > 200\sqrt{p}$)
                Use new PDSYTRD with compiled kernel
            else
                Use unblocked reduction to tridiagonal form (no BLAS)
            endif
        else
            if ($n > 100\sqrt{p}$)
                Use new PDSYTRD with DGEMV
            else
                Use unblocked reduction to tridiagonal form (no BLAS)
            endif
        endif
endif
```

discussion of Line 4.1 below) and we delay the spread of w (which HJS naturally performs at the bottom of the loop) to the top of the loop so that it can be spread in the same message that spreads v.

Our communication pattern has one disadvantage over HJS: it requires redundant computation in the update of w. The discussion of Line 4.1 below explains that we can choose to eliminate this redundant computation by increasing the number of messages.

Line 2.1 in Figure 8.3 In Section 8.4.1 we show how to avoid overflow while using just $2\,n\log(\sqrt{p})$ messages.

Lines 3.2 and 3.6 in Figure 8.3 Only 2 messages are required to transpose a matrix when a square processor layout is used. Each processor, (a, b) must sends a message to, and receive a message from, its transpose processor (b, a). The required time is:

$$\sum_{n'=1}^{n} 2(\alpha + 2\,n'\,\beta) = 2\,n\,\alpha + 2\,n^2\,\beta$$

Line 4.1 in Figure 8.3 $w = w - W\,V^T v - V\,W^T v$ can be computed in a number of ways. W, V and v are distributed across processor rows and replicated across proces-

Figure 8.3: Execution time model for the new PDSYTRD. Line numbers match Figure 4.5(PDSYTRD execution time) where possible.

		computation		communication	
		overhead	imbalance	latency	bandwidth
	do $ii = 1, n, \mathsf{nb}$				
	$mxi = \min(ii + \mathsf{nb}, n)$				
	do $i = ii, mxi$				
Update current (i^{th}) column of A					
1.2	$A = A - W\,V^T - V\,W^T$				
Compute reflector					
2.1	$v = \text{house}(A)$			$2\,n\,\mathsf{lg}(\sqrt{p})\,\alpha$	
Perform matrix-vector multiply					
1.1, 3.1	spread v, w across			$n\,\mathsf{lg}(\sqrt{p})\,\alpha$	$\frac{n^2\,\mathsf{lg}(\sqrt{p})}{\sqrt{p}}\,\beta$
3.2	transpose v, w				$2\,\frac{n^2}{\sqrt{p}}\,\beta$
3.3	$w = \text{tril}(A)v$;		$\frac{2}{3}\frac{n^3}{p}\gamma_2$		
	$w^T = \text{tril}(A, -1)v^T$				$2\,\frac{n^2}{\sqrt{p}}\,\beta$
Update the matrix-vector product					
4.1	$w = w - W\,V^T v - V\,W^T v$				$\frac{n^2}{\sqrt{p}}\,\beta$
3.6	$w = w + \text{transpose } w^T$				
Compute companion update vector					
5.1	$c = w \cdot v^T$;			$2n\,\mathsf{lg}(\sqrt{p})\,\alpha$	
	$w = \tau\,w - (c\,\tau/2)\,v$				
	end do $i = ii, mxi$				
Perform rank $2k$ update					
6.3	$A = A - W\,V^T - V\,W^T$		$\frac{2}{3}\frac{n^3}{p}\gamma_3$		
	end do $ii = 1, n, \mathsf{nb}$				

sor columns. W^T, V^T and v^T are distributed across processor columns and replicated across processor rows. Furthermore, since only the partial sums contributing to w are known, the updates to w can be made on any processor column, and even spread across various processor columns. Appendix B.1 how this update is performed without communication and shows that there are a range of options which trade off communication and load imbalance.

Line 1.1 in Figure 8.3 updates the current block column. This can be implemented in several ways. LAPACK's DSYTRD uses a right looking update[3] because a matrix-matrix multiply is more efficient than an outer product update. HJS uses a left looking update because on their cyclic data layout, the left looking update allows all processors to be involved, reducing load imbalance.

Line 5.1 in Figure 8.3 Computing $c = w\, v^T$ requires summing c within a processor column. In order to compute w in Line 5.1, c must be known throughout a processor column. To allow w and v to be broadcast in the same message (Line 3.1), c is summed and broadcast in the column that owns column $i+1$ of the matrix.

Line 6.1 in Figure 8.3 No communication is required here. W, V^T and W^T are already replicated as necessary.

8.3 Making the ScaLAPACK symmetric eigensolver easier to use

The next ScaLAPACK symmetric eigensolver will separate internal data layout from external data layout while executing 50% faster than PDSYEVX on a large range of problem sizes on most distributed memory parallel computers and requiring less memory. Separating internal and external data layout allows the user to choose whatever data layout is most appropriate for the rest of their code and to use that data layout regardless of the problem size and computer they are using. Separating internal and external data layouts also makes it easy for the ScaLAPACK symmetric eigensolver to add support for additional data layouts. However, while these ease-of-use issues are the most important advantages of separating internal and external data, we will focus further discussion on how this separation improves performance.

8.4 Details in reducing the execution time of the ScaLAPACK symmetric eigensolver

Separating internal and external data layout will improve the performance of PDSYEVX by allowing PDSYEVX to use different data layouts for different tasks, and by allow-

[3] A right looking update updates the current column with a matrix-matrix multiply. A left looking update updates every column in the block column with an outer product update.

ing PDSYEVX to concentrate only on the most efficient data layout for each task. A reduction to tridiagonal form which only works on a cyclic data layout on a square processor grid will not only have lower overhead and load imbalance than the present reduction to tridiagonal form, but will be able to incorporate techniques that would be prohibitively complicated if they were implemented in a code that must support all data layouts.

Significant reduction of the execution time in PDSYEVX, the ScaLAPACK symmetric eigensolver, requires that all four sources of inefficiency (message latency, message transmission, software overhead and load imbalance) be reduced. Fortunately, as Hendrickson, Jessup and Smith[91] have shown, all of these can be reduced. PDSYEVX sends 3 times as many messages as necessary[4], and require 3 times as much message volume as well[5]. Overhead and load imbalance costs are harder to quantify. Load imbalance costs will be reduced by using data layouts appropriate to each task[6]. If necessary, load imbalance costs can be further reduced at the expense of increasing the number of messages sent. Overhead will be reduced by eliminating the PBLAS, reducing the number of calls to the BLAS and, where a sufficiently good compiler is available, eliminating the calls to the BLAS entirely.

8.4.1 Avoiding overflow and underflow during computation of the Householder vector without added messages

Overflow and underflow can be avoided during the computation of the Householder vector without added messages by using the *pdnrm2* routine to broadcast values. The easiest way to compute the norm of a vector in parallel is to sum the squares of the elements. However, this will lead to overflow if the square of one of the elements or one of the intermediate values are greater than the overflow threshold (likewise underflow occurs if one or more of the squares of the elements or the intermediate vallues is less than the underflow threshold). The ScaLAPACK routine *pdnrm2* avoids underflow and overflow during reductions by computing the norm directly leaving the result on all processors in the processor column. The requires $2\lg(p_r)\alpha$ execution time. In PDSYTRD, $\alpha = A(i{+}1, i)$ is broadcast

[4] PDSYEVX uses $17\, n \log(\sqrt{p})$, HJS uses $9\, n \log(\sqrt{p})$, we will show that this can be reduced to $5\, n \log(\sqrt{p})$ but do not claim that this is minimal.

[5] PDSYEVX sends $(5 \log(\sqrt{p}) + 2)\, n^2/\sqrt{p}$ elements per processor and HJS reduces this to $(\frac{1}{2} \log(\sqrt{p}) + \frac{5}{2})\, n^2/\sqrt{p}$ elements per processor. The design I suggest requires $(\frac{3}{2}\log(\sqrt{p}) + \frac{5}{2}) n^2/\sqrt{p}$ elements per processor but requires fewer messages.

[6] Statically balancing the number of eigenvectors assigned to each processor column will reduce load imbalance in back transformation. Using a smaller block size will reduce load imbalance in reduction to tridiagonal form

to all processors in the processor column, this requires $2\lg(p_r)\alpha$ execution time. In HJS, they sum the squares of the elements and broadcast $\alpha = A(i+1,i)$ at the same time by summing an additional value in the reduction. All processors except for the processor that owns $A(i+1,i)$ contribute 0 to the sum while the processor owning $A(i+1,i)$ contributes $A(i+1,i)$.

In the new PDSYEVX, we will employ this trick, to broadcast α at the same time as the norm is computed. It is slightly more complicated because norm computations do not preserve negative numbers. Hence, we compute two norms: $\max(0,\alpha)$ and $\max(0,-\alpha)$, from these α is easily recovered. Ideally, we need a new PBLAS or BLACS routine which would simultaneously compute a norm and broadcast both it and other values.

8.4.2 Reducing communications costs

Communications costs can be reduced in both reduction to tridiagonal form and back transformation but by vastly different methods. PDSYTRD, ScaLAPACK's reduction to tridiagonal form code, will use a cyclic data layout on a square processor grid to simplify the code, allowing PDSYEVX to use the techniques demonstrated by Hendrickson, Jessup and Smith[91]: direct transpose, a column replicated/row distributed data layout for intermediate matrices and combining messages. In addition, PDSYTRD will delay the last operation in the loop to combine it with the first, reducing the number of messages per loop iteration from 6 to 5.

Communication costs will be reduced in back transformation by using a rectangular grid and a relatively large block size. Most of the communication in back transformation is within processor columns, and the communication within processor columns cannot be pipelined (meaning that it grows as $\log(p_r)$), hence setting p_c to be substantially larger (roughly 4-8 times larger) than p_r will cut message volume nearly in half compared to the message volume required for a square processor grid.

Communications cost could be reduced further on select computers by writing machine specific BLACS implementations[7] , but I don't think that the benefit will justify the

[7] Karp et al.[107] proved that a broadcast or reduction of k elements on p_x processors can be executed in $\log(p_x)\,\alpha + k\,\beta$. Equally importantly, the latency term can be reduced significantly by machine specific code because latency is primarily a software cost, the actual hardware latency is typically less than one tenth of the total observed latency. I believe that by coding broadcasts and reductions in a machine specific manner, I could reduce the latency to ($\alpha_{software} + \log(p_x)\,\alpha_{hardware}$. It might be possible to achieve a similar result using active messages. Machine specific optimization of the BLACS broadcast and reduction codes is attractive because it would benefit all of the ScaLAPACK matrix transformation codes. However,

cost. In PDSYEVX as shipped in version 1.5 of ScaLAPACK, software overhead and load imbalance are roughly twice as high as communications cost on the PARAGON. The new PDSYEVX should reduce communications by at least a factor of 2, and though I hope it will reduce software overhead and load imbalance by close to a factor of 4, overhead and load imbalance will probably remain larger than communications cost. The fact that communication costs is not the dominant factor limiting efficiency limits the improvement that one can expect from machine specific BLACS implementations.

Communications cost in back transformation could be reduced further by overlapping communication and computation and/or using an all-to-all broadcast pattern instead of a series of broadcasts. Back transformation enjoys the luxury of being able to compute the majority of what it needs to communicate in advance. This allows many possibilities for reducing the communications bandwidth cost. The fact that message latency, load imbalance and software overhead costs are modest in back transformation means that a reduction in the communications bandwidth cost ought to result in significant performance improvement in back transformation. However, overlapping communication and computation has historically offered less benefit than in practice than in theory, (see section B.1.6) so I approach this with caution and will not pursue it without first convincing myself that the benefit is significant on several platforms.

8.4.3 Reducing load imbalance costs

Load imbalance can be reduced in both reduction to tridiagonal form and back transformation by careful selection of the block size. The number of messages in reduction to tridiagonal form is not dependent on the data layout block size, hence a cyclic data layout (i.e. block size of 1) will be used, reducing load imbalance. The fact that only half of the flops in reduction to tridiagonal form are BLAS3 flops and the large number of load imbalanced row operations combine to make the optimal algorithmic block size for reduction to tridiagonal form small.

Load imbalance is minimized in back transformation by choosing a block size which assigns a nearly equal number of eigenvectors to each column of processors ($\mathsf{nb} = \lceil n/(k\, p_c) \rceil$ for some small integer k). A block cyclic data layout reduces execution time in back transformation by reducing the number of messages sent, hence we must look for

---

purely from the point of view of improving the performance of the ScaLAPACK symmetric eigensolver this effort probably would not be worth the effort.

other ways to reduce load imbalance. Fortunately, all eigenvectors must be updated at each step, hence a good static load balance of eigenvectors across processor columns eliminates most of the load imbalance in back transformation. The load imbalance within each column of processors is less important because the number of processor rows will be small. The computation of T can be performed simultaneously on all processor columns, eliminating the load imbalance in that step.

8.4.4 Reducing software overhead costs

There are many ways to reduce software overhead, but software overhead is poorly understood and hence it is hard to predict which method will be best. Hendrickson, Jessup and Smith[91] showed that using a cyclic data layout and a square processor grid reduces the number of DTRMV calls from $O(n^2/\mathtt{nb})$ to $O(n)$ because each local matrix is triangular. Using lightweight (no error checking, minimal overhead) BLAS would reduce software overhead, but these are still in the planning stages. If the compiler produces efficient code for a simple doubly nested loop, software overhead can be further reduced by using a compiled code instead of calls to the BLAS. Peter Strazdins has shown that software overhead within the PBLAS can be reduced up to 50%[161, 160]. Alternatively, eliminating the PBLAS entirely would eliminate the overhead associated with the PBLAS. I would prefer to reduce the PBLAS overhead and continue to use the PBLAS. But, that is likely to be much harder than simply abandoning the PBLAS.

When PDSYTRD, ScaLAPACK's reduction to tridiagonal form, was written the PBLAS did not support column-replicated/row-distributed matrices or algorithmic blocking. Hence, many of the ideas mentioned here for improving the performance of PDSYTRD were not available to a PBLAS-based code. PBLAS version 2 now offers these capabilities.

Software overhead cannot be measured separate from other costs and is hence difficult to measure, understand and reason about. It varies widely from machine to machine and can change just by changing the order in which subroutines are linked. We do not, for example, know how much can be attributed to subroutine calls, how much is caused by error checking, how much is caused by loop unrolling and how much is caused by code cache misses.

A good compiler should be able to compute the local portion of Av faster than two calls to DTRMV because a simple doubly nested loop could access each element in the local portion of A only once whereas two calls to DTRMV would require that each element in A be read twice. The result is that the ratio of flops to main memory reads is 4-to-1 in the doubly nested loop versus 2-to-1 in DTRMV[8]. Furthermore, a compiled kernel would avoid the BLAS overhead and might involve less loop unrolling - reducing overhead directly and reducing code cache pressure as well. However, compiler technology is uneven, so we would make using compiled code instead of the BLAS optional.

Unblocked reduction to tridiagonal form will likely be faster than blocked reduction to tridiagonal form on problem sizes where software overhead is the dominant cost. Unblocked reduction to tridiagonal form on a cyclic data layout eliminates load imbalance, requires a minimum of communication and software overhead. The only disadvantage is that all of the $4/3\,n^3$ flops are BLAS2 flops. However, with a good compiler, these BLAS2 flops can perform well on most computers. The kernel in an unblocked reduction to tridiagonal form involves 8 flops to each read-modify-write memory access[9]. Most computers have adequate main memory bandwidth to handle this at full speed. However, not all compilers are good enough yet.

8.5 Separating internal and external data layout without increasing memory usage

Separating internal and external data layout will require memory-intensive data redistribution, but making the data redistribution codes more space efficient will save enough memory space to offset the memory needs of separating internal and external data layout. Data redistributions between two data layouts with different values of p_r, p_c or nb use messages of $O(n^2/(p^{3/2}) + \mathsf{nb}^2)$ data elements. However, degenerate data redistributions between two data layouts with the same values of p_r, p_c or nb use messages of roughly n^2/p elements. In order to avoid treating degenerate data redistributions separately, the current redistribution codes require n^2/p buffer space for all redistributions. Splitting one large message into several smaller ones is not conceptually difficult but will require that the code be rewritten and the testing will have to be augmented to properly exercise the new paths. However, the execution time will not be significantly affected. Both PDLARED2D, the

[8]These ratios are 8-to-1 and 4-to-1 respectively for Hermitian matrices.

[9]The ratio for reducing Hermitian matrices to tridiagonal form is 16 flops per read-modify-=write operation.

eigenvector redistribution routine, and DGMR2D, the general purpose redistribution routine, will have to be modified.

If the redistribution routines are not modified as described above, memory usage would increase from $4n^2/p$ to $6n^2/p$, and run a remote risk of causing the eigensolver to crash. While both PDLARED2D and DGMR2D require n^2/p space and could use the same space, they do not. PDLARED2D uses space passed to it in the WORK array, while DGMR2D calls malloc to allocate space. The eigensolver could crash if a message of n^2/p elements were sent, and the communication system was unable to allocate a buffer of that size. Messages of that size are not required during normal ScaLAPACK eigensolver tests, hence the eigensolver could crash during regular use even after passing all tests and after months or even years of flawless service. Modifying the redistribution routines as we propose, eliminates this potential problem.

Memory needs could be reduced from $4n^2/p$ to $3n^2/p$ by using the space allocated to the input matrix, A, and the output matrix, Z, as internal workspace. This would require a modification to the present calling sequence, probably in the form of a new data descriptor. However, reducing memory usage by 25% may not justify a change to the calling sequence.

Chapter 9

Advice to symmetric eigensolver users

Parallel dense tridiagonal eigensolvers should be used if none of the following counter indications hold. Use a serial eigensolver if the problem is small enough to fit[1]. Use a sparse eigensolver if your input matrix is sparse[2] and you don't need all the eigenvalues or if the matrix is dense and you only need a small fraction of the eigenvalues. Use a Jacobi eigensolver if you need to compute small eigenvalues of a scaled diagonally dominant matrix (or a matrix satidying one of the other properties described by Demmel et al.[56]) accurately.Use a Jacobi eigensolver for small ($n < 100\sqrt{p}$) spectrally diagonally dominant matrices[3].

Currently the three most readily available parallel dense symmetric eigensolvers are `PeIGs` and `ScaLAPACK`'s `PDSYEV` and `PDSYEVX`. `PeIGs` and `PDSYEV` maintain orthogonality among eigenvectors associated with clustered eigenvalues. `PeIGs` and `PDSYEVX` are faster than `PDSYEV`. `PDSYEVX` scales better than either `PeIGs` or `PDSYEV`.

The choice between `PeIGs` and `ScaLAPACK` is probably more a matter of which infrastructure[4] is preferred and is out of the scope of this thesis. Furthermore, it is likely that `PeIGs` will at some point use the `ScaLAPACK` symmetric eigensolver in the future. Hence,

[1] i.e. if memory allows

[2] The break-even point is not known, so I suggest that if your matrix is less than 10% non-zero and you need less than 10% of the eigenvalues you should use a sparse eigensolver.

[3] Spectrally diagonally dominant means that the eigenvector matrix, or a permutation thereof, is diagonally dominant.

[4] `PeIGs` is built on top of Global Arrays[101] while `ScaLAPACK` is built on the `BLACS` or `MPI`.

the upgrade path for both may end up with the same underlying code. If you are not likely to use more than 32 processors, `PeIGs` performance should be acceptable[5]. If your input matrices do not include large clusters of eigenvalues or if you can accept non-orthogonal eigenvectors, `PDSYEVX` is the right choice. Otherwise, i.e. if your input matrix has large clusters of eigenvalues for which you need orthogonal eigenvectors, and you wish to use more than 32 processors, `PDSYEV` is the right choice. Eventually, the imporved version of `PDSYEVX` described in Chapter 8 will be the method of choice in all cases.

[5]Since `PeIGs` uses a 1D data layout, its performance will degrade if you use more than 32 processors.

Part II

Second Part

Bibliography

[1] R.C. Agarwal, S.M. Balle, F.G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5), 1995. also available as:`http://www.almaden.ibm.com/journal/rd/agarw/agarw.html`.

[2] R.J. Allan and I.J. Bush. Parallel diagonalisation routines. Technical report, The CCLRC HPCI Centre at Daresbury Labaratory, 1996. `http://www.dl.ac.uk/TCSC/Subjects/Parallel_Algorithms/diags/diags.doc`.

[3] A. Anderson, D. Culler, D. Patterson, and the NOW Team. A case for networks of workstations: NOW. *IEEE Micro*, Feb 1995. `http://now.CS.Berkeley.EDU/Papers2`.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide (second edition)*. SIAM, Philadelphia, 1995. 324 pages.

[5] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. Computer Science Dept. Technical Report CS-90-105, University of Tennessee, Knoxville, 1990. LAPACK Working Note #20 `http://www.netlib.org/lapack/lawns/lawn20.ps`.

[6] E. Anderson and J. Dongarra. Evaluating block algorithm variants in LAPACK. Computer Science Dept. Technical Report CS-90-103, University of Tennessee, Knoxville, 1990. (LAPACK Working Note #19).

[7] ANSI/IEEE, New York. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[8] P. Arbenz, K. Gates, and Ch. Sprenger. A parallel implementation of the symmetric tridiagonal qr algorithm. In *Frontier's 92, McLEan, Virginia*, 1992.

[9] P. Arbenz and I. Slapničar. On an implementation of a one-sided block jacobi method on a distributed memory computer. *Z. Angew. Math. Mech.*, (76, Suppl. 1):343–344, 1996. `http://www.inf.ethz.ch/personal/arbenz/ICIAM_jacobi.ps.gz`.

[10] Peter Arbenz and Michael Oettli. Block implementations of the symmetric qr and jacobi algorithms. Technical Report 178, Swiss Institue of Technology, 1995. `ftp://ftp.inf.ethz.ch:/pub/publications/tech-reports/1xx/178.ps`.

[11] K. Asanovic. Ipm: Interval performance monitoring. `http://www.icsi.berkeley.edu/~krste/ipm/IPM.html`.

[12] C. Ashcraft. A taxonomy of distributed dense LU factorization methods. Technical Report ECA-TR-161, Boeing Computer Services, March 1991.

[13] Z. Bai and J. Demmel. On a block implementation of Hessenberg multishift QR iteration. *International Journal of High Speed Computing*, 1(1):97–112, 1989. (also LAPACK Working Note #8 `http://www.netlib.org/lapack/lawns/lawn8.ps`).

[14] R. Barlow, D. Evans, and J. Shanehchi. Parallel multisection applied to the eigenvalue problem. *Comput. J.*, 6:6–9, 1983.

[15] R.H. Barlow and D.J. Evans. A parallel organization of the bisection algorithm. *The Computer Journal*, 22(3), 1978.

[16] Mike Barnett, Lance Shuler, Robert van de Geijn, Satya Gupta, David Payne, and Jerrell Watts. Interprocessor collective communication library (InterCom). In *Proceedings of the Scalable High Performance Computing Conference*, pages 357–364. IEEE, 1994. `ftp://ftp.cs.utexas.edu/pub/rvdg/shpcc.ps`.

[17] A. Basermann and P. Weidner. A parallel algorithm for determining all eigenvalues of large real symmetric tridiagonal matrices. *Parallel Computing*, 18:1129–1141, 1992.

[18] K. Bathe. *Finite Element Procedures in Enginerring Analysis.* Prentice Hall, Inc., Englewood Cliffs, NJ, 1982.

[19] 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.

[20] H. Bernstein and M. Goldstein. Parallel implementation of bisection for the calculation of eigenvalues of a tridiagonal symmetric matrices. Technical report, Courant Institute, New York, NY, 1985.

[21] M. Berry and A. Sameh. Parallel algorithms for the singular value and dense symmetric eigenvalues problems. *J. Comput. and Appl. Math.*, 27:191–213, 1989.

[22] Allan J. Beveridge. A general atomic and molecular electronic structure system. available as:`http://gserv1.dl.ac.uk/CFS/gamess_4.html`.

[23] J. Bilmes, K. Asanovic, J. Demmel, D. Lam, and C.-W. Chin. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. Computer Science Dept. Technical Report CS-96-326, University of Tennessee, Knoxville, May 1996. LAPACK Working Note #111 `http://www.netlib.org/lapack/lawns/lawn111.ps`.

[24] C. Bischof and X. Sun. A framework for symmateric band reduction and tridiagonalization. Technical report, Supercomputing Research Center, 1991. (Prism Working Note #3 `ftp://ftp.super.org/pub/prism/wn3.ps`).

[25] C. Bischof, X. Sun, and B. Lang. Parallel tridiagonalization through two-step band reduction. In *Scalable High-Performance Computing Conference.* IEEE Computer Society Press, May 1994. (Also Prism Working Note #17 `ftp://ftp.super.org/pub/prism/wn17.ps`).

[26] C. Bischof, X. Sun, A. Tsao, and T. Turnbull. A study of the invariant subspace decomposition algorithm for banded symmetric matrices. In *Proceedings of the Fifth SIAM Conference on Applied Linear Algebra.* IEEE Computer Society Press, June 1994. (Also Prism Working Note #16 `ftp://ftp.super.org/pub/prism/wn16.ps`).

[27] C. Bischof and C. Van Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Statist. Comput.*, 8:s2–s13, 1987.

[28] Christian Bischof, William Gerorge, Steven Huss-Lederman, Xiaobai Sun, Anna Tsao, and Thomas Turnbull. Prism software, 1997. `http://www.mcs.anl.gov/Projects/PRISM/lib/software.html`.

[29] Christian Bischof, William Gerorge, Steven Huss-Lederman, Xiaobai Sun, Anna Tsao, and Thomas Turnbull. *SYISDA User's Guide*, version 2.0 edition, 1995. `ftp://ftp.super.org/pub/prism/UsersGuide.ps`.

[30] R. H. Bisseling and J. G. G. van de Vorst. Parallel LU decomposition on a transputer network. In G. A. van Zee and J. G. G. van de Vorst, editors, *Lecture Notes in Computer Science, Number 384*, pages 61–77. Springer-Verlag, 1989.

[31] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997. `http://www.netlib.org/scalapack/slug/scalapack_slug.html`.

[32] Jerry Bolen, Arlin Davis, Bill Dazey, Satya Gupta, Greg Henry, David Robboy, Guy Shiffler, David Scott, Mark Stallcup, Amir Taraghi, Stephen Wheat, LeeAnn Fisk, Gabi Istrail, Chu Jong, Roff Riesen, and Lance Shuler. Massively parallel distributed computing: World's first 281 gigaflop supercomputer. In *Intel Supercomputer User's Group*, 1995.

[33] R. P. Brent. *Algorithms for minimization without derivatives.* Prentice-Hall, 1973.

[34] K. Bromley and J. Speiser. Signal processing algorithms, architectures and applications. In *Proceedings SPIE 27th Annual International Technical Symposium*, 1983. Tutorial 31.

[35] S. Carr and R. Lehoucq. Compiler blockability of dense matrix factorizations. *ACM TOMS*, 1977. also available as:`ftp://info.mcs.anl.gov/pub/tech_reports/lehoucq/block.ps.Z`.

[36] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the benefits of mixed data and task parallelism. In Symposium on Parallel Algorithms and Architectures (SPAA), July 1995. http://HTTP.CS.Berkeley.EDU/~yelick/soumen/mixed-spaa95.ps.

[37] H. Chang, S. Utku, M. Salama, and D. Rapp. A parallel Householder tridiagonalization strategem using scattered row decomposition. *I. J. Num. Meth. Eng.*, 26:857–874, 1988.

[38] H.Y. Chang and M.Salama. A parallel Householder tridiagonalization stratagem using scattered square decomposition. *Parallel Computing*, 6:297–312, 1988.

[39] S. Chinchalkar. Computing eigenvalues and eigenvectors of a dense real symmetric matrix on the ncube 6400. Technical Report CTC91TR74, Advanced Computing research Institute, June 1991.

[40] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - Design issues and performance. Computer Science Dept. Technical Report CS-95-283, University of Tennessee, Knoxville, March 1995. LAPACK Working Note #95 http://www.netlib.org/lapack/lawns/lawn95.ps.

[41] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. Computer Science Dept. Technical Report CS-95-292, University of Tennessee, Knoxville, May 1995. LAPACK Working Note #100 http://www.netlib.org/lapack/lawns/lawn100.ps.

[42] 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 http://www.netlib.org/lapack/lawns/lawn55.ps.

[43] C-C Chou, Y. Deng, G. Li, and Y. Wang. Parallelizing strassen's method for matrix multiplication on distributed-memory mimd architectures. *International Journal of Computers and Mathematics with Applications*, 30:45–69, 1995.

[44] Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, and Robert A. van de Geijn. Parallel implementation of blas: General techniques for level 3 blas. Technical Report TR-95-40, Department of Computer Sciences, University of Texas, October 1995. PLAPACK Working Note #4, to appear in Concurrency: Practice and Experience. `http://www.cs.utexas.edu/users/plapack/plawns.html`.

[45] M. Chu. A note on the homotopy method for linear algebraic eigenvalue problems. *Lin. Alg. Appl*, 105:225–236, 1988.

[46] John M. Conroy and Louis J. Podrazik. A parallel inertia method for finding eigenvalues on vector and simd architectures. *SIAM Journal on Statistical Computing*, 16:500–505, March 1995.

[47] F. J. Corbato. On the coding of jacobi's method for computing eigenvalues and eigenvectors of real symetric matrices. *Journal of the ACM*, 10(2):123–125, 1963.

[48] Jessup E.R. Crivelli, S. The cost of eigenvalue computation on distributed memory mimd multiprocessors. *Parallel Computing*, 21:401–422, 1995.

[49] J. Cullum and R. A. Willoughby. *Lanczos algorithms for large symmetric eigenvalue computations*. Birkhaüser, Basel, 1985. Vol.1, Theory, Vol.2. Program.

[50] J.J.M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36:177–195, 1981.

[51] M. Dayde, I. Duff, and A. Petitet. A Parallel Block Implementation of Level 3 BLAS for MIMD Vector Processors. *ACM Transactions on Mathematical Software*, 20(2):178–193, 1994.

[52] E. D'Azevedo. personal communication, 1997. `http://www.epm.ornl.gov/~efdazedo/`.

[53] J. Demmel. CS 267 Course Notes: Applications of Parallel Processing. Computer Science Division, University of California, 1991. 130 pages.

[54] J. Demmel, I. Dhillon, and H. Ren. On the correctness of some bisection-like parallel eigenvalue algorithms in floating point arithmetic. *Electronic Trans. Num. Anal.*, 3:116–140, December 1995. LAPACK working note 70.

[55] J. Demmel, J. J. Dongarra, S. Hammarling, S. Ostrouchov, and K. Stanley. The dangers of heterogeneous network computing: Heterogenous networks considered harmful. In *Proceedings Heterogeneous Computing Workshop '96*, pages 64–71. IEEE Computer Society Press, 1996.

[56] J. Demmel, M. Gu, S. Eisenstat, I. Slapnicar, K. Veselic, and Z. Drmac. Computing the singular value decomposition with high relative accuracy. Computer Science Dept. Technical Report CS-97-348, University of Tennessee, Feb 1997. LAPACK Working Note #2 `http://www.netlib.org/lapack/lawns/lawn2.ps`.

[57] J. Demmel and K. Stanley. The performance of finding eigenvalues and eigenvectors of dense symmetric matrices on distributed memory computers. In *Proceedings of the Seventh SIAM Conference on Parallel Proceesing for Scientific Computing*. SIAM, 1994.

[58] J. Demmel and K. Veselić. Jacobi's method is more accurate than QR. *SIAM J. Mat. Anal. Appl.*, 13(4):1204–1246, 1992. (also LAPACK Working Note #15).

[59] Inderjit Dhillon. *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eogenvector Problem*. PhD thesis, University of California at Berkeley, 1997.

[60] Stéphane Domas and Françoise Tisseur. Parallel implementation of a symmetric eigensolver based on the yau and lu method. In *International Journal of Supercomputer Applications (proceedings of Environments and Tools For Parallel Scientific Computing III, Faverges de la Tour, France, 21-23 August*, 1996.

[61] J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, PA, 1979.

[62] 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.

[63] J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Su broutines. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[64] J. Dongarra, S. Hammarling, and D. Sorensen. Block reduction of matrices to condensed forms for eigenvalue computations. *J. Comput. Appl. Math.*, 27:215–227, 1989. LAPACK Working Note #2 `http://www.netlib.org/lapack/lawns/lawn2.ps`.

[65] J. Dongarra, R. Hempel, A. Hay, and D. Walker. A proposal for a user-level message passing interface in a distributed memory environment. Technical Report ORNL/TM-12231, Oak Ridge National Laboratory, Oak Ridge, TN, February 1993.

[66] J. Dongarra and D. Sorensen. A fully parallel algorithm for the symmetric eigenproblem. *SIAM J. Sci. Stat. Comput.*, 8(2):139–154, March 1987.

[67] J. Dongarra and R. van de Geijn. Reduction to condensed form for the eigenvalue problem on distributed memory computers. Computer Science Dept. Technical Report CS-91-130, University of Tennessee, Knoxville, 1991. LAPACK Working Note #30 `http://www.netlib.org/lapack/lawns/lawn30.ps` also Parallel Computing.

[68] 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.

[69] J. Dongarra and R. C. Whaley. A user's guide to the blacs v1.1. Technical report, University of Tennessee, Knoxville, March 1995. LAPACK Working Note #94 `http://www.netlib.org/lapack/lawns/lawn94.ps`.

[70] C. C. Douglas, M. Heroux, G. Slishman, and R.M. Smith. Gemmw: A portable level 3 blas winograd variant of strassen's matrix-matrix multiply algorithm. *Journal of Computational Physics*, 110:1–10, 1994.

[71] Zlatko Drmač and Krešimir Veselić. Iterative refinement of the symmetric eigensolution. Technical report, University of Colorado at Boulder, 1997. zlatko@cs.colorado.edu.

[72] P.J. Eberlein and M. Mantharam. Jacobi sets for the eigenproblem and their effect of convergence studied by graphci representations. Technical report, SUNY Buffalo, 1990.

[73] P.J. Eberlein and M. Mantharam. New jacobi for parallel computations. *Parallel Computing*, 19:437–454, 1993.

[74] G. Fann and R. Littlefield. Performance of a fully parallel dense real symmetric eigensolver in quantum chemistry applications. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computation*. SIAM, 1994.

[75] G. Fann and R. J. Littlefield. A parallel algorithm for householder tridiagonalization. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 409–413. SIAM, 1993.

[76] R. Fellers. Performance of `pdsyev`, ... Mathematics Dept. Master's Thesis available by anonymous ftp to `http://cs-tr.CS.Berkeley.EDU/NCSTRL/`, University of California, 1997.

[77] V. Fernando, B. Parlett, and I. Dhillon. A way to find the most redundant equation in a tridiagonal system. Berkeley Mathematics Dept. Preprint, 1995.

[78] UTK Joint Institute for Computational Science, 1997. `http://www-jics.cs.utk.edu/SP2/sp2_config.html`.

[79] J.G.F. Francis. The QR transformation: A unitary analogue to the LR transformation, parts I and II. *The Computer Journal*, 4:265–272, 332–345, 1961.

[80] K. Gates. A rank-two divide and conquer method for the symmetric tridiagonal eigenproblem. In *Frontier's 92, McLean, Virginia*, 1992.

[81] Kevin Gates. Using inverse iteration to improve the divide and conquer algorithm. Technical Report 159, Swiss Institue of Technology, 1991.

[82] Kevin Gates and Peter Arbenz. Parallel divide and conquer algorithms for the symmetric tridiagonal eigenproblem. Technical Report 222, Swiss Institue of Technology, 1995. `ftp://ftp.inf.ethz.ch:/pub/publications/tech-reports/2xx/222.ps`.

[83] W. Givens. Numerical computation of the characteristic values of a real matrix. Technical Report 1574, Oak Ridge National Laboratory, 1954.

[84] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 1983.

[85] J. Götze. On the parallel implementation of jacobi and kogbetliantz algorithms. *SIAM J. on Sci. Comput.*, pages 1331–1348, 1994. http://www.nws.e-technik.tu-muenchen.de/~jugo/pub/SIAMjac.ps.Z.

[86] A. Greenbaum and J. Dongarra. Experiments with QL/QR methods for the symmetric tridiagonal eigenproblem. Computer Science Dept. Technical Report CS-89-92, University of Tennessee, Knoxville, 1989. LAPACK Working Note #17 http://www.netlib.org/lapack/lawns/lawn17.ps.

[87] Numerical Algorithms Group, 1997. http://www.nag.co.uk/numeric.html.

[88] M. Gu and S. Eisenstat. A stable algorithm for the rank-1 modification of the symmetric eigenproblem. Computer Science Dept. Report YALEU/DCS/RR-916, Yale University, September 1992.

[89] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM J. Mat. Anal. Appl.*, 16(1):172–191, January 1995.

[90] M. Hegland, M. H. Kahn, and Osborne M. R. A parallel algorithm for the reduction to tridiagonal form for eigendecomposition. Technical Report TR-CS-96-06, Australian National University, 1996. http://cs.anu.edu.au/techreports/1996/index.html.

[91] B. Hendrickson, E. Jessup, and C. Smith. A parallel eigensolver for dense symmetric matrices. Technical Report SAND96–0822, Sandia National Labs, Albuquerque, NM, March 1996. Submitted to SIAM J. Sci. Comput.

[92] G. Henry. personal communication, 1997. http://www.cs.utk.edu/~ghenry/.

[93] Greg Henry. *Improving Data Re-Use in Eigenvalue-Related Computations*. PhD thesis, Cornell University, 1994.

[94] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Draft, January 1993. Also available as technical report CRPC-TR 92225, Center for Research on Parallel Computation, Rice University.

[95] Y. Huo and R. Schreiber. Efficient, massively parallel eigenvalue computations. preprint, 1993.

[96] S. Huss-Lederman, Jacobson E.M., J. R. Johnson, Tsao A., and T. Turnbull. "strassen's algorithm for matrix multiplication: Modeling, analysis, and implementation". Technical report, Center for Computing Sciences, 1996. (Also Prism Working Note #34 `ftp://ftp.super.org/pub/prism/wn34.ps`).

[97] S. Huss-Lederman, A. Tsao, and G. Zhang. "a parallel implementation of the invariant subspace decomposition algorithm for dense symmetric matrices". In *Proceedings of Sixth SIAM conference on Parallel Processing for Scientific Computing*, March 1993. (Also Prism Working Note #9 `ftp://ftp.super.org/pub/prism/wn9.ps`).

[98] IBM, Kingston, NY. *Engineering and Scientific Subroutine Library — Guide and Reference*, release 3 edition, 1988. Order No. SC23-0184.

[99] I. Ipsen and E. Jessup. Solving the symmetric tridiagonal eigenvalue problem on the hypercube. *SIAM J. Sci. Stat. Comput.*, 11(2):203–230, 1990.

[100] C.G.J. Jacobi. Über ein leichtes verfahren die in der theorie der säculärstörungen vorkommenden gleichungen numerisch aufzulösen. *Crelle's Journal*, 30:51–94, 1846.

[101] Pacific Northwest Labaratories Jarek Nieplocha, 1996. http://www.emsl.pnl.gov:2080/docs/global/ga.html.

[102] E. Jessup and I. Ipsen. Improving the accuracy of inverse iteration. *SIAM J. Sci. Stat. Comput.*, 13(2):550–572, 1992.

[103] B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. Report UMINF-95.18, Department of Computing Science, Umeå University, S-901 87 Umeå, Sweden, 1995. *To appear in ACM Trans. Math. Software* LAPACK Working Note #107 `http://www.netlib.org/lapack/lawns/lawn107.ps`.

[104] B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS: Portability and Optimization Issues . Technical report, Department of Computing Science, Umeå University, 1997. *To appear in ACM Trans. Math. Software.*

[105] W. Kahan. Accurate eigenvalues of a symmetric tridiagonal matrix. Computer Science Dept. Technical Report CS41, Stanford University, Stanford, CA, July 1966 (revised June 1968).

[106] R.K. Kamilla, X.G. Wu, and J.K. Jain. Composite fermion theory of collective excitations in fractional quantum hall effect. *Physical Review Letters*, 1996.

[107] 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.

[108] L. Kaufman. Banded eigenvalue solvers on vector machines. *ACM Trans. Math. Soft.*, 10:73–86, 1984.

[109] L. Kaufman. A parallel qr algorithm for the symmetric tridiagonal eigenvalue problem. *Journal of Parallel and Distributed Computing*, 23:429–434, 1994.

[110] D. Koebel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, 1994.

[111] A. S. Krishnakumar and M. Morf. Eigenvalues of a symmetric tridiagonal matrix: A divide and conquer approach. *Numer. Math.*, 48:349–368, 1986.

[112] Peter Freche Krystian Pracz, Martin Janssen. Correlation of eigenstates in the critical regime of quantum hall systems. *J. Phys. Condens. Matter*, 8:7147–7159, 1996. also available as:`http://xxx.lanl.gov/abs/cond-mat/9605012`.

[113] D. Kuck and A. Sameh. A parallel QR algorithm for symmetric tridiagonal matrices. *IEEE Trans. Computers*, C-26(2), 1977.

[114] J.R. Kuttler and V.G. Sigillito. Eigenvalues of the laplacian in two dimensions. *SIAM Review*, 26:163–193, 1984.

[115] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[116] B. Lang. A parallel algorithm for reducing symmetric banded matrices to tridiagonal form. *SIAM J. Sci. Comput.*, 14(6), November 1993.

[117] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[118] Thomas J. LeBlanc and Evangelos P. Markatos. Shared memory vs. message passing in shared-memory multiprocessors. In *4th Symp. on Parallel and Distributed Processing*, 1992. `ftp://ftp.cs.rochester.edu/pub/papers/systems/92.ICPP.locality_vs_load_balancing.ps.Z`.

[119] R. B. Lehoucq. Arpack software. `http://www.mcs.anl.gov/home/lehoucq/software.html`.

[120] K. Li and T.-Y. Li. An algorithm for symmetric tridiagonal eigenproblems — divide and conquer with homotopy continuation. *SIAM J. Sci. Comp.*, 14(3), May 1993.

[121] Rencang Li and Huan Ren. An efficient tridiagonal eigenvalue solver on cm 5 with laguerre's iteration. Computer Science Division CSD-94-848, University of California, 1994. `http://sunsite.berkeley.edu/Dienst/UI/2.0/Describe/ncstrl.ucb%2fCSD-94-848`.

[122] T.-Y. Li and Z. Zeng. Laguerre's iteration in solving the symmetric tridiagonal eigenproblem - a revisit. Michigan State University preprint, 1992.

[123] T.-Y. Li, H. Zhang, and X. H. Sun. Parallel homotopy algorithm for symmetric tridiagonal eigenvalue problems. *SIAM J. Sci. Stat. Comput.*, 12:464–485, 1991.

[124] W. Lichtenstein and S. L. Johnsson. Block cyclic dense linear algebra. *SIAM J. Sci. Comp.*, 14(6), November 1993.

[125] R.J. Littlefield and K. J. Maschhoff. Investigating the performance of parallel eigensolvers for large processor counts. *Theretica Chemica Acta*, 84:457–473, 1993.

[126] S.-S. Lo, B. Phillipe, and A. Sameh. A multiprocessor algorithm for the symmetric eigenproblem. *SIAM J. Sci. Stat. Comput.*, 8(2):155–165, March 1987.

[127] Mi Lu and Xiangzhen Qiao. Applying parallel computer systems to solve symmetric tridiagonal eigenvalue problems. *Parallel Computing*, 18:1301–1315, 1992.

[128] S. C. Ma, M. Patrick, and D. Szyld. A parallel, hybrid algorithm for the generalized eigenproblem. In Garry Rodrigue, editor, *Parallel Processing for Scientific Computing*, chapter 16, pages 82–86. SIAM, 1989.

[129] R. S. Martin, C. Reinsch, and J. H. Wilkinson. Householder's tridiagonalization of a symmetric matrix. *Numerische Mathematik*, 11:181–195, 1968.

[130] K. Maschhoff. Parpack software. `http://www.caam.rice.edu/~kristyn/parpack_home.html`.

[131] R. Mathias. The instability of parallel prefix matrix multiplication. *SIAM J. Sci. Stat. Comput.*, 16(4):956–973, July 1995.

[132] Gary Oas. Universal cubic eigenvalue repulsion for random normal matrices. *Physical Review E*, 1996. also available as:`http://xxx.lanl.gov/abs/cond-mat/9610073`.

[133] David C. O'Neal and Raghurama Reddy. Solving symmetric eigenvalue problems on distributed memory machines. In *Proceedings of the Cray User's Group*, pages 76–96. Cray Inc., 1994.

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

[135] B. Parlett. *Acta Numerica*, chapter The new qd algorithms, pages 459–491. Cambridge University Press, 1995.

[136] B. Parlett. The construction of orthogonal eigenvectors for tight clusters by use of submatrices. Center for Pure and Applied Mathematics PAM-664, University of California, Berkeley, CA, January 1996. submitted to SIMAX.

[137] B. Parlett. personal communication, 1997.

[138] B. N. Parlett. Laguerre's method applied to the matrix eigenvalue problem. *Mathematics of Computation*, 18:464–485, 1964.

[139] B.N. Parlett and I.S. Dhillon. On Fernando's method to find the most redundant equation in a tridiagonal system. *Linear Algebra and its Applications*, 267:247–279, 1997. Nov.

[140] Antoine Petitet. *Algorithmic Redistribution Methods for Block Cyclic Decompositions.* PhD thesis, University of Tennessee, 1996.

[141] C. P. Potter. A parallel divide and conquer eigensolver. `http://sawww.epfl.ch/SIC/SA/publications/SCR95/7-95-27a.html`.

[142] M. Pourzandi and B. Tourancheau. A parallel performance study of jacobi-like eigenvalue solution. `http://www.netlib.org/tennessee/ut-cs-94-226.ps`.

[143] Earl Prohofsky. *Statistcal Mechanics and Stability of Macromolecules*. Cambridge University Press, 1995.

[144] B. Putnam, E. W. Prohofsky, K. C. Lu, and L. L. Van Zandt. Breathing modes and induced resonant melting of the double helix. *Physics Letters*, 70A, 1979.

[145] C. Reinsch. A stable rational qr algorithm for the computation of the eigenvalues of an hermitian, tridiagonal matrix. *Num. Math.*, 25:591–597, 1971.

[146] H. Ren. *On error analysis and implementation of some eigenvalue and singular value algorithms*. PhD thesis, University of California at Berkeley, 1996.

[147] J. Rutter. A serial implementation of Cuppen's divide and conquer algorithm for the symmetric eigenvalue problem. Mathematics Dept. Master's Thesis `http://sunsite.berkeley.edu/Dienst/UI/2.0/Describe/ncstrl.ucb%2fCSD-94-799`, University of California, 1991.

[148] R. Saavedra, W. Mao, D. Park, J. Chame, and S. Moon. The combined effectiveness of unimodular transformations, tiling, and software prefetching. In *Proceedings of the 10th International Parallel Processing Symposium*. IEEE Computer Society, April 15–19 1996.

[149] V. Sarkar. Automatic selection of high order transformations in the IBM ASTU Optimizer. IBM Software Solutions Division Report, 1996.

[150] R. Schreiber. Solving eigenvalue and singular value problems on an undersized systolic array. *SIAM J. Sci. Stat. Comput.*, 7:441–451, 1986.

[151] D. Scott, M. Heath, and R. Ward. Parallel block Jacobi eigenvalue algorithms using systolic arrays. *Lin. Alg. & Appl.*, 77:345–355, 1986.

[152] G. Seifert, Th. Heine, O. Knospe, and R. Schmidt. Computer simulations for the structure and dynamics of large molecules, clusters and solids. In *Lecture Notes in Computer Science*, volume 1067, page 393. Springer-Verlag, 1996.

[153] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1976.

[154] C. Smith, B. Hendrickson, and E. Jessup. A parallel algorithm for householder tridiagonalization. In *Proceedings of the Fifth SIAM Conference on Applied Linear Algebra*, pages 361–365. SIAM, 1994.

[155] D. Sorensen and P. Tang. On the orthogonality of eigenvectors computed by divide-and-conquer techniques. *SIAM J. Num. Anal.*, 28(6):1752–1775, 1991.

[156] J. Speiser and H. Whitehouse. Parallel processing algorithms and architectures for real time processing. In *Proceedings SPIE Real Time Signal Processing IV*, 1981.

[157] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–355, 1969.

[158] P. Strazdins. Matrix factorization using distributed panels on the fujitsu ap1000. In *IEEE First International Conference on Algorithms And Architectures for Parallel Processing*, Brisbsane, April 1995. `http://cs.anu.edu.au/people/Peter.Strazdins/papers.html#DBLAS`.

[159] P. Strazdins. A high performance, portable distributed blas implementation. In *Fifth Parallel Computing Workshop for the Fujitsu PCRF*, Kawasaki, November 1996. `http://cs.anu.edu.au/people/Peter.Strazdins/papers.html#DBLAS`.

[160] P. Strazdins. personal communication, 1997. `http://cs.anu.edu.au/people/Peter.Strazdins`.

[161] P. Strazdins. Reducing software overheads in parallel linear algebra libraries. Technical report, Australian National University, 1997. Submitted to PART'97, The 4th Annual Australasian Conference on Parallel And Real-Time Systems, 29 - 30 September 1997, The University of Newcastle, Newcastle, Australia.

[162] P. Swarztrauber. A parallel algorithm for computing the eigenvalues of a symmetric tridiagonal matrix. To appear in *Math. Comp.*, 1993.

[163] D. Szyld. Criteria for combining inverse iteration and Rayleigh quotient iteration. *SIAM J. Num. Anal.*, 25(6):1369–1375, December 1988.

[164] Thinking Machines Corporation. *CMSSL for CM Fortran: CM-5 Edition, version 3.1*, 1993.

[165] S. Toledo. Locality of reference in lu decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18-4, 1997. `http://theory.lcs.mit.edu/~sivan/029774.ps.gz`.

[166] Alessandro De Vita, Giulia Galli, Andrew Canning, and Roberto Car. A microscopic model for surface-induced graphite-to-diamond transitions. *Nature*, 379, Feb 8 1996.

[167] D. Watkins. *Fundamentals of Matrix Computations*. Wiley, 1991.

[168] R. Whaley. Automatically tunable linear algebra subroutines, 1997. `http://www.netlib.org/utk/projects/atlas`.

[169] 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 `http://www.netlib.org/lapack/lawns/lawn73.ps`.

[170] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, Oxford, 1965.

[171] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, Shih-Wei Liao, C. Tseng, Mary W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. HTML from http://suif.stanford.edu/suif/suif-overview/suif.html.

[172] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.

[173] Y.-J. J. Wu, A. A. Alpatov, C. Bischof, and R. A. van de Geijn. "a parallel implementation of symmetric band reduction using plapack". In *Scalable Parallel Library Conference*, 1996. (Also Prism Working Note #35 `ftp://ftp.super.org/pub/prism/wn35.ps`).

[174] Shing-Tung Yau and Ya Yan Lu. Reducing the symmetric matrix eigenvalue problem to matrix multiplications. *SIAM J. Sci. Comput.*, 14(1):121–136, January 1993.

[175] Paul G. Hipes Yi-Shuen Mark Wu, Steven A. Cuccaro and Aron Kuppermann. Quantum-mechanical reactive scattering using a high-performance distributed-memory parallel computer. *Chem. Phys. Lett.*, 168:429–440, 1990.

Appendix A

Variables and abbreviations

Table A.1: Variable names and their uses

Name	**Meaning**
(a, b)	The processor in processor row a and processor column b.
A	The input matrix (partially reduced).
$A(i, j)$	The i, j element in the (partially reduced) matrix A.
c	The number of eigenvalues in the largest cluster of eigenvalues.
C	The set of all processor columns.
c_a	The current processor column within the sub-grid.
c_b	The current processor column sub-grid.
e	The number of eigenvalues required.
j	The current column, $A(j : n, j : n)$ being the un-reduced portion of the matrix.
j'	The column within the current block column, $j' = \mod(j, \mathsf{nb})$
$\mathsf{lg}(\sqrt{p})$	$\log_2 \sqrt{p}$
m	The number of eigenvectors required.
mb	The row block size. Used only when we discuss rectangular blocks. In general, the row block size and column block size are assumed to be equal and are written as nb.
mullen	A compile time parameter in the `PBLAS` which controls the panel size used in `PBLAS` symmetric matrix vector multiply routine, `PDSYMV`.
n	The size of the input matrix A.
nb	The blocking factor. In `PDSYEVX` the data layout and algorithmic blocking factor are the same. In `HJS` the data layout blocking factor is 1 and nb refers to the algorithmic blocking factor.
p	The number of processors used in the computation.
pbf	Panel blocking factor. The panel width used in `DGEMV` in `PDSYEVX` and `DGEMM` in `PDSYEVX` and `HJS` is $\mathsf{pbf} \times \mathsf{nb}$.
p_r	The number of processor rows in the process grid.
p_{r1}	The number of processor rows in a sub-grid.
p_{r2}	The number of processor sub-grid rows.
p_c	The number of processor columns in the process grid.
p_{c1}	The number of processor columns in a sub-grid.
p_{c2}	The number of processor sub-grid columns.
R	The set of all processor rows.
r_a	The current processor row within the sub-grid.
r_b	The current processor row sub-grid.
spread	In a "spread across", every processor in current processor column broadcasts to every other processor in the same processor row. In a "spread down", every processor in current processor row broadcasts to every other processor in the same processor column.
$\mathrm{tril}(A, 0)$	The lower triangular part, including the diagonal, of the unreduced part of the input matrix A, i.e. $A(j : n, j : n)$
$\mathrm{tril}(A, -1)$	The lower triangular part, excluding the diagonal, of the unreduced part of the input matrix A, i.e. $A(j : n, j : n)$

Table A.2: Variable names and their uses (continued)

Name	**Meaning**
v	The vector portion of the householder reflector.
V	The current column of householder reflectors. Size: $n-j+j'$ by j'.
$V(j-j':n,1:j')$	The current column of householder reflectors. Size: $n-j+j'$ by j'.
vnb	The imbalance in the 2D block-cyclic distribution of the eigenvector matrix.
w	The companion update vector. i.e. the vector used in $A = A - vw^T - Wv^T$ to reduce A
W	The current column of companion update vectors. Size: $n-j+j'$ by j'.
$W(j-j':n,1:j')$	The current column of companion update vectors. Size: $n-j+j'$ by j'.

Abbreviation	Meaning
CPU	Central Processing Unit
FPU	Floating Point Unit

Table A.3: Abbreviations

Symbol	Meaning	Terms included
α	The message initiation cost for `BLACS` send and receive.	$n\,\mathrm{lg}(p)$,n
β	The inverse bandwidth cost for `BLACS` send and receive.	$\frac{n^2\ \mathrm{lg}(p)}{\sqrt{p}}$, $\frac{n^2}{\sqrt{p}}$, $n \times \mathsf{nb}\ \mathrm{lg}(p)$
δ_3	`DGEMM` (matrix-matrix multiply) subroutine overhead plus the time penalty associated of invoking `DGEMM` on small matrices.	$\frac{n^2}{\mathsf{nb}^2 \times \mathsf{pbf}}$, $\frac{n}{\mathsf{nb}}$
γ_3	Time required per `DGEMM` (matrix-matrix multiply) flop.	$\frac{n^3}{p}$, $\frac{n^2 \times \mathsf{nb}}{\sqrt{p}}$
δ_2	`DGEMV` (matrix-vector multiply) subroutine overhead plus the time penalty associated of invoking `DGEMV` on small matrices.	n
γ_2	Time required per `DGEMV` (matrix-vector multiply) flop.	$\frac{n^3}{p}$, $\frac{n^2 \times \mathsf{nb}}{\sqrt{p}}$
$\gamma_\div$	Time required per divide.	$\frac{n^2}{p}$,n
$\gamma_\surd$	Time required per square root.	
γ_1	Time required per `BLAS1` (scalar-vector) flop.	$\frac{n^2}{p}$,n
δ_1	Subroutine overhead for `BLAS1` and similar codes.	$\frac{n^2}{\sqrt{p}}$
δ_4	Subroutine overhead for the `PBLAS`.	n

Table A.4: Model costs

Appendix B

Further details

B.1 Updating v during reduction to tridiagonal form

Line 4.1, $w = w - W\,V^T v - V\,W^T v$ in Figure 8.3 can be computed with minimal communication, minimal computation or with an intermediate amount of both communication and computation. Indeed, Line 4.1 can be computed with $O((\frac{n^2}{p} + n^2 \frac{\mathsf{nb}}{p^r})\gamma_2 + n \log(p^{r-0.5})\alpha)$ cost for various $r \in [0.5, 1.0]$. $r = 1.0$ corresponds to the minimal computation cost option (discussed in section B.1.3) while $r = 0.5$ corresponds to the minimal (zero) communication cost option (discussed in section B.1.2). Section B.1.4 describes the intermediate options in a generalized form which includes both the minimum communication and minimum computation options as special cases.

The plethora of options for the update of v stems from the fact that the input matrices W, V, WT and VT are replicated across the relevant processors while the input/output vector v is stored as partial sums across the processor columns in each of the processor rows. The input matrices are replicated because they will need to be replicated later to update A. The vector v is stored as partial sums because that is how it is initially computed, and because the combine operation used to compute v from the partial sums has not been performed at this point.

Throughout this section we only discuss computing WV^Tv. VW^Tv can be computed in a similar manner. Moreover, the two computations, and all associated communication, can be merged to reduce software overhead and message latency costs.

B.1.1 Notation

In describing most parallel linear algebra codes, including all codes in this thesis outside of this appendix, we need not explicitly state the processor on which a value is stored. $A_{i,j}$ is understood to live on the processor that owns row i and column j. The nb' element array tmp contains different values on different processors. Therefore, for the discussion in this appendix, an additional subscript is added to tmp to indicate the processor column. Furthermore, some entries in tmp are left undefined at various stages, therefore we use $j \in \{c_a\}$ to indicate all columns j owned by processor column c_a. i.e. $\mathsf{tmp}_{j\in\{c_a\},c_a} = val$ means that $\forall j \in \{c_a\}$, tmp_j on processor c_a is assigned val. For extra clarity within a display we write this as $\underset{j\in\{c_a\}}{\mathsf{tmp}_{j,c_a}}$.

B.1.2 Updating v without added communication

Line 4.1, $w = w - W\,V^T v - V\,W^T v$ in Figure 8.3 can be computed without any communication other than that needed to compute v without the update. It initially appears that $w = w - W \cdot V^T v - V \cdot W^T v$ requires communication because computing $\mathsf{tmp} = V^T v$ requires summing nb' values[1] within each processor column, and computing $w = w - W \cdot \mathsf{tmp}$ requires that tmp be broadcast within each processor column. However, $W \cdot V^T v$ can be computed with a single sum within each processor row, and by delaying the sum needed to compute w, one of them can be avoided completely. Figure B.1 derives how $W \cdot V^T v$ can be computed with a single sum within each processor row.

Line 3 The transformation from line 2 to line 3 is the standard way that a matrix vector multiply is performed in parallel. The leftmost sum is the local portion, the middle sum is the sum over all processors in the processor column.

Line 4 Delay the sum over all processors in the processor column until after multiplying by W. The rightmost two sums involve only local values.

Figure B.2 shows how to compute $W \cdot V^T v$ without added communication.

Line 5 Local computation of $VT \cdot v$. Operations:

$$\sum_{i=1,\mathsf{nb}}^{n} \sum_{\mathsf{nb}'=1}^{\mathsf{nb}} 2\,\frac{i}{p_r}\,\mathsf{nb}'\,\gamma_2 = \tfrac{1}{2}\,n^2\,\frac{\mathsf{nb}}{p_r}\,\gamma_2$$

[1] $\mathsf{nb}' = i - ii - 1$ is the number of columns in H

Figure B.1: Avoiding communication in computing $W \cdot V^T v$

$$\mathsf{tmp} = W \cdot V^T v \qquad \text{(Line 1)}$$

$$\mathsf{tmp}_i = \sum_{1 \le j \le \mathsf{nb}'} W_{ij} \sum_{k \in \{C\}} V_{kj} v_k \qquad \text{(Line 2)}$$

$$\mathsf{tmp}_i = \sum_{1 \le j \le \mathsf{nb}'} W_{ij} \sum_{\substack{1 \le R \le p_r \\ k \in \{C\}}} H_{kj} h_k \qquad \text{(Line 3)}$$

$$\mathsf{tmp}_i = \sum_{\substack{C \in p_c \\ 1 \le j \le \mathsf{nb}'}} W_{ij} \sum_{k \in \{C\}} H_{kj} h_k \qquad \text{(Line 4)}$$

Line 6 Local computation of $W \cdot \mathsf{tmp}$. Operations:

$$\sum_{i=1,\mathsf{nb}}^{n} \sum_{\mathsf{nb}'=1}^{\mathsf{nb}} 2 \frac{i}{p_r} \mathsf{nb}' \gamma_2 = \tfrac{1}{2} n^2 \frac{\mathsf{nb}}{p_r} \gamma_2$$

Line 7 Effect of summing res_i within each processor row. This operation is merged with the unavoidable summation of w within each processor row, hence this operation is not performed and has no cost.

B.1.3 Updating w with minimal computation cost

Figure B.3 shows how $W \cdot V^T v$ can be performed with only $O(\frac{n^2}{\sqrt{p}} + \frac{n^2\,\mathsf{nb}}{p})$ computation by distributing the computation of $\mathsf{tmp} = VT \cdot v$ and $w = w + W \cdot \mathsf{tmp}$ over all the processors. Each of the nb columns of VT is assigned to one processor row, hence each processor row is assigned $\frac{\mathsf{nb}}{\sqrt{p}}$ columns of VT. Each processor row computes the portion of $VT \cdot v$ assigned to it, leaving the answer on the diagonal processor in this row. The diagonal processors then broadcast the $\frac{\mathsf{nb}}{\sqrt{p}}$ elements of $VT \cdot v$ which they own to all of the processors within their processor column. Finally, each processor computes $w = w + W \cdot \mathsf{tmp}$ for the values of W and tmp which it owns.

Figure B.2: Computing $W \cdot V^T v$ without added communication

$$\mathsf{tmp}_{j,C} = \sum_{k \in \{C\}} VT_{k,j}\, v_k \qquad \text{(Line 5)}$$

$$\underset{i \in \{R\}}{\mathsf{res}_{i,C}} = \sum_j W_{i,j}\ \mathsf{tmp}_{j,C} \qquad \text{(Line 6)}$$

$$= \sum_j W_{i,j} \sum_{k \in \{C\}} VT_{k,j}\, v_k$$

$$\sum_C \underset{i \in \{R\}}{\mathsf{res}_{i,C}} = \sum_{\substack{j \\ 1 \le C \le p_c}} W_{i,j} \sum_{k \in \{C\}} VT_{k,j}\, v_k \qquad \text{(Line 7)}$$

$$= \sum_j W_{i,j} \sum_k VT_{k,j}\, v_k$$

Line 8 Local computation of $VT \cdot v$. Operations:

$$\sum_{i=1,\mathsf{nb}}^{n} \sum_{\mathsf{nb}'=1}^{\mathsf{nb}} 2\, \frac{i}{p_r}\, \frac{\mathsf{nb}'}{p_c}\, \gamma_2 = \tfrac{1}{2}\, \frac{n^2\, \mathsf{nb}}{p}\, \gamma_2$$

Line 9 Combine $\mathsf{tmp}_{j \in \{R\},C}$ within each processor column, leaving the answer on the diagonal processor. Operations:

$$\sum_{i=1,\mathsf{nb}}^{n} \sum_{\mathsf{nb}'=1}^{\mathsf{nb}} \log(p_c)\, (\alpha + \frac{\mathsf{nb}'}{p_c}\, \beta) = n\, \log(p_c)\alpha + \tfrac{1}{2}\, \frac{n\, \mathsf{nb}}{p_c}\, \log(p_c)\beta$$

Line 10 Broadcast $\mathsf{tmp}_{j \in \{C\},C}$ within each processor row from the diagonal processor. Operations:

$$\sum_{i=1,\mathsf{nb}}^{n} \sum_{\mathsf{nb}'=1}^{\mathsf{nb}} \log(p_c)\, (\alpha + \frac{\mathsf{nb}'}{p_c}\, \beta) = n\, \log(p_c)\alpha + \tfrac{1}{2}\, \frac{n\, \mathsf{nb}}{p_c}\, \log(p_c)\beta$$

Line 11 Local computation of $W \cdot \mathsf{tmp}$. Operations:

$$\sum_{i=1,\mathsf{nb}}^{n} \sum_{\mathsf{nb}'=1}^{\mathsf{nb}} 2\, \frac{i}{p_r}\, \frac{\mathsf{nb}'}{p_c}\, \gamma_2 = \tfrac{1}{2}\, \frac{n^2\, \mathsf{nb}}{p}\, \gamma_2$$

Figure B.3: Computing $W \cdot V^T v$ with minimal computation

$$\underset{j\in\{R\}}{\mathsf{tmp}_{j,C}} = \sum_{k\in\{C\}} VT_{k,j}\, v_k \qquad \text{(Line 8)}$$

$$\begin{aligned} \forall_{R=C}\ \underset{j\in\{C\}}{\mathsf{tmp}_{j,C}} &= \sum_{\substack{1\le c_l\le p_c \\ k\in\{c_l\}}} VT_{k,j}\, v_k \qquad \text{(Line 9)} \\ &= \sum_{k} VT_{k,j}\, v_k \end{aligned}$$

$$\underset{j\in\{C\}}{\mathsf{tmp}_{j,C}} = \sum_{k\in\{C\}} VT_{k,j}\, v_k \qquad \text{(Line 10)}$$

$$\begin{aligned} \underset{i\in\{R\}}{\mathsf{res}_{i,C}} &= \sum_{j\in\{C\}} W_{i,j}\ \underset{j\in\{C\}}{\mathsf{tmp}_{j,C}} \qquad \text{(Line 11)} \\ &= \sum_{j\in\{C\}} W_{i,j} \sum_{k\in\{C\}} VT_{k,j}\, v_k \end{aligned}$$

$$\begin{aligned} \sum_{C} \underset{i\in\{R\}}{\mathsf{res}_{i,C}} &= \sum_{\substack{1\le C\le p_c \\ j\in\{C\}}} W_{i,j} \sum_{k\in\{C\}} VT_{k,j}\, v_k \qquad \text{(Line 12)} \\ &= \sum_{j} W_{i,j} \sum_{k} VT_{k,j}\, v_k \end{aligned}$$

Line 12 Effect of summing res_i within each processor row. This operation is merged with the unavoidable summation of w within each processor row, hence this operation is not performed and has no cost.

The update of w in HJS requires similar communication and computation costs although the patterns of communication are quite different. HJS uses recursive halving to spread the result of $\mathsf{tmp} = V^T v$, computes $W \cdot \mathsf{tmp}$ on all processors, and uses recursive doubling to compute w while simultaneously spreading it to all processor columns. Although the `BLACS` do not offer recursive halving and recursive doubling operations we could build them out of `BLACS` sends and receives but that incurs higher latency costs.

B.1.4 Updating w with minimal total cost

Line 4.1, $w = w - W W^T w - W W^T w$ in Figure 8.3 can be computed with $O(\frac{n^2 \mathsf{nb}}{p^r}\gamma_2 + n \log(p^{r-0.5})\alpha)$ cost for any $r \geq 0.5$. On a high latency machine, one can reduce the total number of messages by increasing the load imbalance. On a low latency machine, one can reduce the load imbalance by using more messages. The two options described in the preceding sections are special cases of the general case of methods described in this section. Section B.1.2 corresponds to $r = 0.5$. Section B.1.3 corresponds to $r = 1.0$.

This method has not been implemented and hence has not been proven to result in decreased execution times in practice.

Methods corresponding to $0.5 < r < 1.0$ require what amounts to a four dimensional processor grid. The $p_r \times p_c$ processor grid is divided into $p_{r2} \times p_{c2}$ sub-grids with each sub-grid consisting of $p_{r1} \times p_{c1}$ processors. We restrict our attention to square processor grids and square processor sub-grids, hence $p_r = p_c, p_{r1} = p_{c1}$ and $p_{r2} = p_{c2}$. Each processor column is identified by a pair of numbers, (c_a, c_b), s.t. $1 \leq c_a \leq p_{c1}$ and $1 \leq c_b \leq p_{c2}$. Likewise, each processor row is identified by a pair of numbers, (r_a, r_b), s.t. $1 \leq r_a \leq p_{r1}$ and $1 \leq r_b \leq p_{r2}$. No modifications are needed to the `BLACS` to support this method because each processor belongs to only two 2 dimensional processor grids: the normal two dimensional data layout and a two dimensional data layout containing only those processors in the same processor sub-grid, i.e. with the same r_b and c_b.

Figure B.4 shows the general method for updating w using a 4 dimensional data layout. The nb' elements of tmp are distributed over the p_{r1} processor rows and columns within each processor block, such that each processor row and column owns roughly $\frac{\mathsf{nb}'}{p_{r1}}$

elements of tmp.

Figure B.4: Computing $W \cdot V^T v$ on a four dimensional processor grid

$$\underset{j\in\{r_a\}}{\mathsf{tmp}_{j,(c_a,c_b)}} = \sum_{k\in\{(c_a,c_b)\}} VT_{k,j}\, v_k \qquad \text{(Line 13)}$$

$$\forall_{r_a=c_a} \underset{j\in\{c_a\}}{\mathsf{tmp}_{j,(c_a,c_b)}} = \sum_{\substack{1\le c_l\le p_{c1}\\ k\in\{(c_a,c_b)\}}} VT_{k,j}\, v_k = \sum_{k\in\{(*,c_b)\}} VT_{k,j}\, v_k \qquad \text{(Line 14)}$$

$$\underset{j\in\{c_a\}}{\mathsf{tmp}_{j,(c_a,c_b)}} = \sum_{k\in\{(c_a,c_b)\}} VT_{k,j}\, v_k \qquad \text{(Line 15)}$$

$$\underset{i\in(r_a,r_b)}{\mathsf{res}_{i,(c_a,c_b)}} = \sum_{j\in\{c_a\}} W_{i,j} \underset{j\in\{c_a\}}{\mathsf{tmp}_{j,(c_a,c_b)}} = \sum_{j\in\{c_a\}} W_{i,j} \sum_{k\in\{(*,c_b)\}} VT_{k,j}\, v_k \qquad \text{(Line 16)}$$

$$\sum_{(c_a,c_b)} \underset{i\in(r_a,r_b)}{\mathsf{res}_{i,(c_a,c_b)}} = \sum_{\substack{1\le c_a\le p_{c1}\\ 1\le c_b\le p_{c2}\\ j\in\{c_a\}}} W_{i,j} \sum_{k\in\{(*,c_b)\}} VT_{k,j}\, v_k = \sum_{\substack{1\le c_a\le p_{c1}\\ j\in\{c_a\}}} W_{i,j} \sum_{\substack{1\le c_b\le p_{c2}\\ k\in\{(*,c_b)\}}} VT_{k,j}\, v_k = \sum_j W_{i,j} \sum_k VT_{k,j}\, v_k \qquad \text{(Line 17)}$$

B.1.5 Notes to figure B.4

Line 13 Local computation of $VT \cdot v$. Operations:

$$\sum_{i=1,\mathsf{nb}}^{n} \sum_{\mathsf{nb}'=1}^{\mathsf{nb}} 2\,\frac{i}{p_r}\,\frac{\mathsf{nb}'}{p_{c1}}\,\gamma_2 = \tfrac{1}{2}\, n^2\, \mathsf{nb}/(p_r\, p_{c1})\,\gamma_2$$

Line 14 Combine $\mathsf{tmp}_{j\in\{r_a\},(c_a,c_b)}$ within each processor sub-grid column, leaving the answer on the diagonal processor (i.e. $r_a = c_a$) within each sub-grid. Operations:

$$\sum_{i=1,\mathsf{nb}}^{n} \sum_{\mathsf{nb}'=1}^{\mathsf{nb}} \log(p_{c1})\,(\alpha + \frac{\mathsf{nb}'}{p_{c1}}\beta) = n\,\log(p_{c1})\alpha + \tfrac{1}{2}\frac{n\,\mathsf{nb}}{p_{c1}}\,\log(p_{c1})\beta$$

Line 15 Broadcast $\mathsf{tmp}_{j\in\{r_a\},(c_a,c_b)}$ within each processor sub-grid row from the diagonal processor in that sub-grid row. Operations:

$$\sum_{i=1,\mathsf{nb}}^{n} \sum_{\mathsf{nb}'=1}^{\mathsf{nb}} \log(p_{c1})\,(\alpha + \frac{\mathsf{nb}'}{p_{c1}}\beta) = n\,\log(p_{c1})\alpha + \tfrac{1}{2}\frac{n\,\mathsf{nb}}{p_{c1}}\,\log(p_{c1})\beta$$

Line 16 Local computation of $W \cdot \mathsf{tmp}$. Operations:

$$\sum_{i=1,\mathsf{nb}}^{n} \sum_{\mathsf{nb}'=1}^{\mathsf{nb}} 2\,\frac{i}{p_r}\,\mathsf{nb}'/p_{c1}\gamma_2 = \tfrac{1}{2}\,\frac{n^2\,\mathsf{nb}}{p_r\,p_{c1}}\gamma_2$$

Line 17 Effect of summing res_i within each processor row. This operation is merged with the unavoidable summation of w within each processor row, hence this operation is not performed and has no cost.

B.1.6 Overlap communication and computation as a last resort

There are numerous studies showing that overlapping communication and computation improves performance, but most of them show only modest improvement. Arbenz and Slapnicar[9] show a 5% improvement by overlapping communication and computation while Pourzandi and Tourancheau show a 6% improvement. Those that show the greatest improvement combine communication and computation overlap with other equally important techniques such as pipelining and lookahead[32].

I don't know why overlapping communication and computation leads to only modest improvements. In theory it ought to hide most of the communication costs. There are several possible explanations, all of which presumably contribute. I suspect that the most important reason for the disappointing savings from overlap is that overhead and not communication costs are not the primary factor limiting efficiency. A second important reason is that most of the cost of communication on todays distributed memory machines is the cost of moving the data between the node and the network, not moving data within the network. The cost of moving data to and from the node always involves main memory cycles, unless the main memory is dual ported (i.e. expensive), which must be stolen from the execution of the rest of the code. Further the latency cost is almost all software overhead, hence during the message setup the cpu is busy and cannot compute.

The disadvantage to communication and computation overlap is that it adds complexity which can be put to better use elsewhere. Both the Pourzandi/Tourancheau and Arbenz/Slapnicar studies used a 1D data layout in Jacobi although a 2D data layout offers lower communication and costs $O(\frac{n^2}{\sqrt{p}})$ versus $O(n^2)$ and lower overhead costs. They would have done better to use a 2D data layout and delayed (potentially forever) consideration of communication and computation overlap.

B.2 Matlab codes

B.2.1 Jacobi

The following is the matlab code for Table 7.4.

```
n = 1000;
p = 64;
blacsalpha = 65.9e-6;
blacsbeta=.146e-6;
dividebeta=3.85e-6;
squarerootbeta=7.7e-6;
blasonebeta=.074e-6;
dgemmalpha=103e-6;
dgemmbeta=.0215e-6;
term(1) = 8 * sqrt(p) * ( log2(p) - 3 ) * blacsalpha
term(2) = 7/2 * n^2 / sqrt(p) * blacsbeta
term(3) = 1/8 * n^2 / sqrt(p) * log2(p) * blacsbeta
term(4) = 1/2 * n^2 / sqrt(p) * dividebeta
term(5) = 1/4 * n^2 / sqrt(p) * squarerootbeta
term(6) = 3/8 * n^3 / p * blasonebeta
term(7) = 8 * sqrt(p) * dgemmalpha
term(8) = 5 * n^3 / p * dgemmbeta
time = sum(term)
```

Appendix C

Miscellaneous matlab codes

C.1 Reduction to tridiagonal form

The following matlab code performs an unblocked reduction to tridiagonal form. It produces the same values, up to roundoff, of D, E and TAU as LAPACK's DSYTRD and ScaLAPACK's PDSYTRD.

```
%
%  tridi - An unblocked, non-syymetric reduction to tridiagonal form
%
%  This file creates an input matrix A, reduces it to tridiagonal form
%  and tests to make sure that the reduction was performed correctly.
%
%  outputs:
%    D, E - The tridiagonal matrix
%    tau
%    A - The lower half holds the householder updates
%
%
%  Produce the input matrix
%
N = 7;
A =  hilb(N) + toeplitz( [ 1 (1:(N-1))*i ] );
B = A;    % Keep a copy to check our work later.

%
%  Reduce to tridiagonal form
%
n = size(A,1);
```

```
I = eye(N);

for j =1:n-1
%
% Compute the householder vector: v
%
  clear v;
  v(1:n,1) = zeros(n,1);
  v(j+1:n,1) = A(j+1:n,j);
  alpha = A(j+1,j);
  beta = - norm(v) * real(alpha) / abs( real(alpha) ) ;

  tau(j) = ( beta - alpha ) / beta ;
  v = v / ( alpha - beta ) ;
  v(j+1) = 1.0 ;

%
% Perform the matrix vector multiply:
%
  w = A * v ;

%
% Compute the companion update vector: w
%
  w = tau(j) * w ;
  c = w' * v;
  w = (w - (c * tau(j) / 2 ) * v );


  D(j) = A(j,j);
  E(j) = beta ;
%
% Updte the trailing matrix
%
  A = A - v * w' - w * v';

%
% Store the household vector back into A
%
  A(j+2:n,j) = v(j+2:n);
end
D(n) = A(n,n);
```

```
%
%  Check to make sure that the reduction was performed correctly.
%
DE = diag(D) + diag(E,-1) + diag(E,1) ;

Q=I;
for j = 1:n-1
  clear house
  house(1:n,1) = zeros(n,1);
  house(j+1:n,1) = A(j+1:n,j);
  house(j+1,1) = 1.0;
  Q =   (I- tau(j)' * house * house') * Q ;
end

norm( B -  Q' * DE * Q )
```