A comparison of the solution of the Symmetric Eigenvalue Problem with ScaLAPACK and Jacobi methods*†

Domingo Giménez ‡

Abstract

In this paper we compare the performance obtained when solving the Symmetric Eigenvalue Problem with ScaLAPACK (routine `pxsyevx`) and with two blocked Jacobi methods on a Paragon. The conclusion is that when using a large number of processors a lower execution time is obtained with a Jacobi method than with `pxsyevx`.

1 Introduction

The Symmetric Eigenvalue Problem appears in a lot of applications in science and engineering. In some cases the problems to solve are of great dimensions [1] and it is necessary to solve them on powerful machines and using highly efficient algorithms.

In the last years LAPACK [2] has become the most popular package for solving this type of problem, on single processors and Shared Memory Multiprocessors.

In order to solve every day bigger problems, whose size depends on the new necessities of the science, it is necessary to obtain algorithms highly efficient and scalable, allowing us to solve those problems on Massively Parallel Processors (MPP). ScaLAPACK [3] is a package for solving Dense Linear Algebra problems on MPPs. It is being developed and the intention is to include in ScaLAPACK all the routines in LAPACK. But programming on Distributed Memory Multicomputers is more difficult than on Shared Memory Multiprocessors, and that is why at present ScaLAPACK has only a few of the routines of LAPACK. The Symmetric Eigenvalue Problem can be solved with the ScaLAPACK routine `pxsyevx`.

Other methods for solving the Symmetric Eigenvalue Problem seem to be more scalable than the method used in ScaLAPACK, and they are promising for MPPs. Some of these methods are Jacobi methods and methods based on invariant subspace decomposition [4, 5].

Highly efficient and scalable Jacobi methods working on MPPs can be designed. On a previous work [6] we designed an algorithm for a logical square mesh of processors, and in this paper a new algorithm on a logical triangular mesh is presented. We compare the performance of these two Jacobi methods and the routine `pxsyevx` of ScaLAPACK. Our purpose is to determine if the Jacobi methods can be competitive on MPPs with other methods (as that used in ScaLAPACK) of lower computational cost. The conclusion is that for medium matrix size and a large number of processors the Jacobi methods have a lower

*Partially supported by Comisión Interministerial de Ciencia y Tecnología, project TIC96-1062-C03-02, and Consejería de Cultura y Educación de Murcia, Dirección General de Universidades, project FI-con 96/9.

†The experiments have been performed on the 512 node Paragon on the CSCC parallel computer system operated by Caltech on behalf of the Concurrent Supercomputing Consortium (access to this facility was provided by the PRISM project).

‡Departamento de Informática y Sistemas. Univ de Murcia. Aptdo 4021. 30001 Murcia. (domingo@dif.um.es)

execution time than the routine of ScaLAPACK. The lowest execution time is obtained with the Jacobi method on a triangular mesh.

In section two we briefly describe the way in which LAPACK, ScaLAPACK and the Jacobi methods solve the Symmetric Eigenvalue Problem. In section three we show and comment on the results of an experimental comparison of the different methods on a Paragon.

2 Different methods for solving the Symmetric Eigenvalue Problem

2.1 The routine `xsyevx` of LAPACK

The Symmetric Eigenvalue Problem can be solved in LAPACK with the routines **`xsyevx`**. These routines work by reducing the symmetric matrix A to tridiagonal form ($QAQ^T = T$), and then computing the eigenvalues (and optionally the eigenvectors) of the tridiagonal matrix T. In that way, the eigenvalues of matrix A are computed with a cost $\frac{4}{3}n^3 + O\left(n^2\right)$.

2.2 The routine `pxsyevx` of ScaLAPACK

The routines **`pxsyevx`** of ScaLAPACK are equivalent to the routines **`xsyevx`** of LAPACK.

In ScaLAPACK the reduction of the initial matrix A to the tridiagonal matrix T is performed with routines of level 3 PBLAS. The theoretical cost of the algorithm is the same as in LAPACK, and the parallelism is exploited by the use of PBLAS.

The execution time of the routine is ([7]):

$$(1) \qquad k_3 \frac{2}{3}\frac{n^3}{p} + k_2\left(\frac{2}{3}\frac{n^3}{p} + 120\frac{n^2}{p}\right) + 60\frac{n^2}{p} + 21\alpha n \log p + 5\beta\frac{n^2}{\sqrt{p}}\log p$$

when computing eigenvalues, and:

$$(2) \qquad k_3 \frac{8}{3}\frac{n^3}{p} + k_2\left(\frac{2}{3}\frac{n^3}{p} + 520\frac{n^2}{p}\right) + 71\frac{n^2}{p} + 21\alpha n \log p + \beta\left(7\frac{n^2}{\sqrt{p}}\log p + 4\frac{n^2}{p}\right)$$

when computing eigenvalues and eigenvectors.

In these formulae, and in the following formulae in this section, k_3, k_2 and k_1 represent the cost of an operation performed using BLAS 3, BLAS 2 or BLAS 1, respectively, and α and β represent the start-up time and the word-sending time of a communication.

2.3 Jacobi methods

The Jacobi method was the most widely used to solve the Symmetric Eigenvalue Problem during approximately a century [8], but its high computational cost (approximately $3n^3 \log n$ when computing eigenvalues, and $6n^3 \log n$ when computing eigenvalues and eigenvectors) resulted in the method falling out of favour in the 50's, when the QR algorithm was developed.

The method has been revived recently due to better stability properties [9] and straightforward parallelization [10, 11]. Because of the inherent parallelism of the Jacobi methods, it is possible to design highly efficient and scalable Jacobi algorithms for MPPs.

On [6] an algorithm by blocks working on a logical square mesh of processors was designed. The blocked algorithm has a higher computational cost than other Jacobi algorithms, but the use of blocks allows us to perform most of the computations with BLAS 3, and with the algorithm by blocks a great reduction on the execution time is obtained in comparison with the algorithms not working by blocks.

In this paper we present a new blocked Jacobi algorithm on a logical triangular mesh of processors. This algorithm presents better properties than the blocked Jacobi method for a logical square mesh when the number of processors increases, and is more suitable for MPPs.

We will briefly analyse these two Jacobi methods and, after that, we will experimentally compare them with the routine `pdsyevx` of ScaLAPACK, in order to analyse whether the Jacobi methods can be competitive (on MPPs and in execution time) with the methods based on reduction to tridiagonal form.

To explain the methods we consider a matrix A of size $n \times n$ divided into blocks of size $t \times t$, and the algorithm will work considering each one of these blocks as elements, and working only with the lower triangular part of the matrix. In figure 1 the logical division in blocks of the matrix and the memory reserved to store the matrix on odd and even steps is represented. In this figure $\frac{n}{t} = 16$ and the rows and columns of blocks are numbered from 0 to 15, and each block of size $t \times t$ is represented by an `x`. These blocks are grouped in bigger blocks, grouping $2k$ consecutive rows and columns of blocks of size $t \times t$ (in the figure $k = 2$).

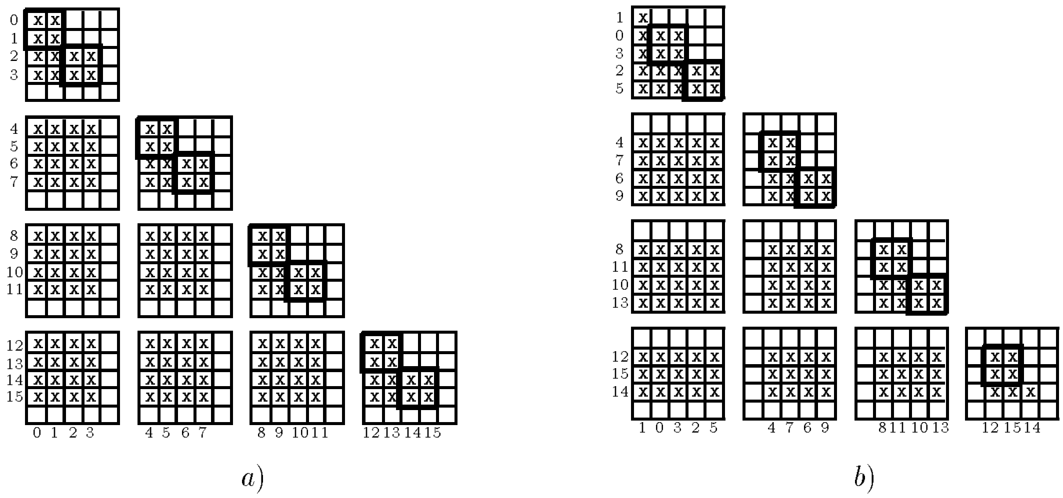

FIG. 1. *Logical distribution of the matrix on odd and even steps.*

After updating the matrix, a movement of rows and columns of blocks is necessary in order to obtain the new grouping of blocks according to the odd-even ordering. This movement generates the logical division and storage scheme of the matrix shown in figure 1 b). We can see that some of the additional memory positions initially reserved are now used.

A set of sweeps are performed over the blocks of size $2t \times 2t$ marked in the figure. But the difference with the sweeps performed in the previous step is that in the first step the sweeps were performed nullifying once each nondiagonal element, and in the successive steps the sweeps are performed nullifying only elements in the subdiagonal blocks of size $t \times t$, because the elements on blocks of size $t \times t$ in the main diagonal of blocks were nullified in the first step.

After the second step, a new movement of rows and columns of blocks is necessary to obtain the new grouping of data. In that way, the blocks are stored again as in figure 1 a).

The work continues in the same way until $\frac{n}{t}$ steps are performed. Then each nondiagonal element on A has been nullified once, and a sweep has been performed in A. The work continues performing successive sweeps until convergence is reached.

Until now we have explained the block Jacobi method by considering the matrix divided into blocks of size $t \times t$ and these blocks grouped in blocks of size $2tk \times 2tk$ considered as units to assign memory. To explain the two parallel methods we have designed, is now an easy task:

To obtain an algorithm for a logical square mesh of processors it suffices to consider the matrix divided in four submatrices of size $\frac{n}{2} \times \frac{n}{2}$ ($A_{11}, A_{12}, A_{21}, A_{22}$) and distribute matrix A_{12} in the system in the obvious way, and the matrices A_{11} and A_{22} folding them over the processors in the system. In this way the processors in the main antidiagonal of processors will store 3 blocks of size $2tk \times 2tk$ (from A_{11}, A_{12} and A_{22}), and the other processors will store 2 blocks (one from A_{12} and another from A_{11} or A_{22}). In this way we have $n = 4t\sqrt{p}k$, with p the number of processors.

To obtain the algorithm on a logical triangular mesh of processors the storage scheme is obvious, and consists in assigning to each processor one of the blocks of size $2tk \times 2tk$. In this case we have $n = 2tkq$, with $q = \frac{-1+\sqrt{1+8p}}{2}$.

In the following we will call the algorithm for a square mesh JACSQ, and the algorithm for a triangular mesh JACTRI.

The movements of data we analysed considering the matrix divided into blocks of size $2tk \times 2tk$ generate the corresponding transference of data between processors in the parallel algorithms. And also a transference of rotations' matrices from processors in the main diagonal (JACTRI) or antidiagonal (JACSQ) of processors to other processors in the same row and column of processors is necessary, in order to communicate to each processor in the system the rotations' matrices needed to update the part of the matrix stored.

When computing only eigenvalues the cost per sweep of JACSQ is approximately:

$$k_3\left(4\frac{n^3}{p}+2\frac{n^2t}{\sqrt{p}}\right)+k_1 12\left(\frac{n^2t}{\sqrt{p}}+\frac{nt^2}{\sqrt{p}}\right)+\alpha\left(4+\sqrt{p}\right)\frac{n}{t}+\beta\left(2+\frac{3}{\sqrt{p}}\right)n^2 \tag{3}$$

and the cost per sweep of JACTRI is approximately:

$$k_3\left(8\frac{n^3}{q^2}+2\frac{n^2t}{q}\right)+k_1 12\left(\frac{n^2t}{q}+\frac{nt^2}{q}\right)+\alpha\left(4+q\right)\frac{n}{t}+\beta\left(2+\frac{6}{q}\right)n^2 \tag{4}$$

with $q = \frac{-1+\sqrt{1+8p}}{2}$. When the number of processors increases $q \simeq \sqrt{2}\sqrt{p}$, and (4) can be approximated by:

$$k_3\left(4\frac{n^3}{p}+\sqrt{2}\frac{n^2t}{\sqrt{p}}\right)+k_1 6\sqrt{2}\left(\frac{n^2t}{\sqrt{p}}+\frac{nt^2}{\sqrt{p}}\right)+\alpha\left(4+\sqrt{2}\sqrt{p}\right)\frac{n}{t}+\beta\left(2+\frac{3\sqrt{2}}{\sqrt{p}}\right)n^2 \tag{5}$$

The eigenvectors can be computed accumulating the rotations' matrices in a matrix Q initialized to the matrix I. Working all the processors in the actualization of Q, the cost per sweep of this updating is $k_3 4\frac{n^3}{p}$ flops, and the cost per sweep of the algorithms when computing eigenvalues and eigenvectors can be obtained adding this expression to (3) and (4).

The cost per sweep of a sequential Jacobi algorithm not working by blocks is $3n^3$ flops when computing eigenvalues, and $6n^3$ flops when computing eigenvalues and eigenvectors, and we could conclude blocking is a bad idea. But the highest order in (3) and (5) is affected by k_3, which represents the cost of an arithmetic operation using BLAS 3; and in the non blocked algorithms the operations are performed using BLAS 1. Thus, it is possible to obtain a big reduction using algorithms by blocks, and this is what happens with these algorithms.

Comparing (1), (3) and (5) we conclude that `pdsyevx` is faster than JACSQ and JACTRI, and the difference in the execution time when n increases depends on the values of k_3 and k_2. For example, if $k_2 = 3k_3$ (which is a reasonable assumption) `pdsyevx` will be approximately 7 times faster than JACSQ and JACTRI. But the Jacobi algorithms have smaller communication times, and it is possible to overlap communications and computations in some parts of the execution, which was not considered when obtaining the theoretical execution times. That allows us to be a bit optimistic about the Jacobi algorithms, and it could be interesting to compare experimentaly `pdsyevx` with JACSQ and JACTRI on MPPs.

Comparing the execution times of JACSQ and JACTRI (expressions (3) and (5)) we see JACTRI has lower arithmetic cost than JACSQ, but a higher communication cost. To decide which algorithm is better we need to compare them experimentaly, but JACTRI is simpler than JACSQ, and this simplicity will produce lower overhead and smaller execution time when the matrix size and the number of processors increases.

3 Experimental comparison

The experiments have been performed on a 512 node Paragon. Each node has an arithmetic processor i860, and an additional processor to handle the communications [12].

It is interesting to compare the Jacobi algorithms with the ScaLAPACK routine, but also to compare these routines with the routine in LAPACK.

In table 1 the execution time (in seconds) obtained with a sequential Jacobi algorithm by blocks, with `pdsyevx` and with `dsyevx` is shown. The tables show results obtained with randomly generated matrices with values between -10 and 10, and similar times are obtained in other cases. Results obtained when computing only eigenvalues, or eigenvalues and eigenvectors, are shown. We can see the LAPACK routine is the quickest when computing only eigenvalues, but when computing eigenvalues and eigenvectors the ScaLAPACK routine is better (for big matrices) due to the use of level 3 PBLAS.

The performance of the routine `pdsyevx` depends on the block size, and a block size of 8 has been chosen to obtain these results. With this block size the execution time is close to the optimum [3], and the difference between the time obtained with the optimum block size and a block size of 8 is insignificant.

Also the performance of the blocked Jacobi algorithm depends on the block size. A block size of 16 has been used to obtain the results shown in the table and the difference between the time obtained with the optimum block size and with a block size of 16 is also insignificant in this case.

The blocked Jacobi method has a cost per sweep of $4n^3$ flops when computing eigenvalues, and not a cost of $3n^3$ flops as in the nonblocked method, but better execution times are obtained with the blocked method and big matrices. Comparing the theoretical execution time of the Jacobi method and the routines of LAPACK and ScaLAPACK, we can see it is a difficult task to develop a Jacobi algorithm competitive with the methods based on reduction to tridiagonal form. But we can be a bit optimistic if we consider that the difference between the Jacobi algorithm and the routines is lower than the theoretical difference: the LAPACK routine is between 7 and 26 (depending on the matrix size) times faster than the Jacobi algorithm when computing only eigenvalues, and between 2 and 4 times faster when computing eigenvalues and eigenvectors, and the ScaLAPACK routine is between 1.2 and 6.4 times faster when computing only eigenvalues, and between 1.3 and 7.1 when computing eigenvalues and eigenvectors.

TABLE 1

Execution time (in seconds) for solving the Symmetric Eigenvalue Problem with different methods.

	eigenvalues			eigenvalues/eigenvectors		
	LAPACK	ScaLAPACK	Jacobi	LAPACK	ScaLAPACK	Jacobi
128	0.83	4.41	5.47	3.26	5.83	8.06
256	2.10	15.37	31.99	16.09	20.40	54.83
384	5.16	33.64	100.92	51.51	45.27	171.46
512	10.35	59.39	229.94	113.27	81.33	406.62
640	17.60	92.97	416.67	221.82	136.44	760.65
768	28.46	135.52	717.95	384.10	234.15	1298.66
896	42.41	187.41	1075.81	626.71	345.73	2053.11
1024	59.13	248.38	1587.57	936.54	456.15	3270.20

The routine `pdsyevx` of SCALAPACK is faster than the Jacobi algorithm on one processor, but we are interested in studying the possibility of obtaining better results with Jacobi methods when using a lot of processors. Table 2 shows the execution time of `pdsyevx` and the blocked Jacobi methods when computing eigenvalues. The table shows the lowest execution times (in seconds) obtained with each method, and the number of processors with which this execution time has been obtained. For each matrix size the smaller execution time is marked, and we can see JACTRI is clearly the best method for medium size matrices, but with bigger matrices `pdsyevx` tends to be the better of the three methods.

TABLE 2

Comparison of the execution time of `pdsyevx` *and two blocked Jacobi methods.*

	`pdsyevx`		JACSQ		JACTRI	
size	time	proc.	time	proc.	time	proc.
128	2.32	10	2.77	4	**1.98**	10
256	3.91	36	5.57	256	**1.98**	136
384	5.69	36	7.16	256	**2.86**	300
512	7.45	64	9.63	256	**6.62**	136
640	9.33	64	8.35	400	**8.30**	210
768	11.40	64	16.57	256	**10.04**	300
896	13.72	64	19.08	256	**11.53**	406
1024	**16.06**	64	25.01	256	27.49	136

To obtain the times of the Jacobi methods we have performed a sweep and considered 10 sweeps are necessary to reach the convergence. Because the number of sweeps to reach the convergence is approximately $\log n$, the comparison is more favourable to Jacobi methods with small matrices and more favourable to `pdsyevx` with big matrices.

The number of processors used with Jacobi methods is restricted because $n = 4tk\sqrt{p}$ in JACSQ and $n = 2tkq$, $q = \frac{-1+\sqrt{1+8p}}{2}$, in JACTRI. This is why the optimum number of processors varies greatly in JACQS and JACTRI. The routine `pdsyevx` has been executed with the same number of processors as JACQS and JACTRI and using a mesh as similar to a square mesh as possible, which produces an execution time near to the optimum [3].

The optimum time is obtained with `pdsyevx` using a small number of processors (36 or 64), and with a greater number of processors the execution time increases. This is due to the low scalability of `pdsyevx`. The Jacobi methods present better scalability, and this is why the optimum time is obtained with JACQS and JACTRI with a bigger number of processors. This bigger scalability allows us to obtain lower execution time with Jacobi methods than with `pdsyevx`, although `pdsyevx` is about six times faster than the Jacobi method in one processor.

To analyse how the different methods work when the number of processors and the matrix' size vary, in table 3 the efficiency of `pdsyevx` is compared with the efficiencies of JACSQ and JACTRI. The efficiencies are obtained comparing each parallel algorithm with itself executed on one processor.

Table 3
Comparison of the efficiency of `pdsyevx` *and the two blocked Jacobi methods.*

proc:	4		16		64		256	
method:	syev	JACSQ	syev	JACSQ	syev	JACSQ	syev	JACSQ
128	0.41	0.51	0.10	0.08	0.02	0.02		
256	0.63	0.68	0.22	0.40	0.05	0.09	0.01	0.02
384	0.72	0.71	0.32	0.46	0.09	0.17	0.02	0.05
512	0.75	0.80	0.39	0.53	0.12	0.30	0.02	0.09
640	0.80	0.80	0.46	0.55	0.15	0.32	0.03	0.12
768	0.82	0.79	0.50	0.58	0.18	0.38	0.04	0.16
896	0.83	0.81	0.53	0.58	0.21	0.40	0.05	0.21
1024	0.85	0.85	0.57	0.68	0.24	0.46	0.06	0.25
proc:	3		10		36		136	
method:	syev	JACTRI	syev	JACTRI	syev	JACTRI	syev	JACTRI
128	0.57	0.53	0.19	0.28	0.04	0.05	0.01	0.01
256	0.78	0.69	0.37	0.49	0.11	0.28	0.02	0.13
384	0.80	0.68	0.45	0.58	0.16	0.42	0.03	0.24
512	0.84	0.72	0.52	0.64	0.21	0.47	0.04	0.26
640	0.86	0.69	0.58	0.65	0.26	0.52	0.06	0.29
768	0.87	0.69	0.62	0.65	0.30	0.54	0.07	0.36
896	0.88	0.69	0.65	0.67	0.34	0.55	0.09	0.35
1024	0.89	0.70	0.67	0.72	0.37	0.62	0.10	0.43

Analysing table 3 we can conclude the Jacobi algorithms are more suitable for parallelization on MPPs than the method used in `pdsyevx`. When the number of processors increases the efficiency of the Jacobi algorithms is clearly greater than the efficiency of `pdsyevx`, and this difference increases with the number of processors. JACTRI has a low efficiency with a small number of processors, which is mainly due to the imbalance in the arithmetic computation. But this imbalance is compensated when using a large number of processors with the overlap of arithmetic operations and communications, and so, JACTRI is the method with the best efficiency when the number of processors increases, which makes this method the most suitable for MPPs.

4 Conclusions

For solving the Symmetric Eigenvalue Problem computing all the eigenvalues of a dense matrix the Jacobi methods can be competitive on MPPs with other methods based on reduction of the original matrix to condensed form (as the method used in the routine `pdsyevx` of ScaLAPACK).

We have compared two Jacobi algorithms with `pdsyevx` on a 512 nodes Paragon, and the experiments confirm that the Jacobi methods are more scalable and so more suitable for MPPs. Of these two Jacobi algorithms, the algorithm we introduce in this paper (an algorithm for a logical triangular mesh of processors) is the one with the best scalability properties, and with this algorithm lower execution times than with `pdsyevx` are obtained when solving problems of dimension of some hundreds using a large number of processors. The experiments allow us to hope that for bigger matrices the Jacobi method can also be better than `pdsyevx`, but with a greater number of processors.

The programmes we have designed are not good programmes, and have been designed only for research proposal, and they have restrictions in the size of the matrix and the mesh. What we propose is to design a good program using the algorithm JACTRI and include it in ScaLAPACK.

References

[1] A. Edelman. Large dense linear algebra in 1993: The parallel computing influence. *The International Journal of Supercomputer Applications*, 7(2):113–128, 1993.

[2] 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.* SIAM, 1992.

[3] ScaLAPACK User's Guide. 1996.

[4] L. Auslander and A. Tsao. On parallelizable eigensolvers. *Ad. App. Math.*, 13:253–261, 1992.

[5] C. H. Bischof, W. George, S. Huss-Lederman, X. Sun, A. Tsao and T. Turnbull. SYISDA Users' Guide. A Parallel Eigensolver Based on a Symmetric Invariant Subspace Decomposition Algorithm. Technical report, Argonne National Laboratory, 1995.

[6] D. Giménez, V. Hernández, R. van de Geijn and A. M. Vidal. A Jacobi method by blocks on a mesh of processors. *To appear in Concurrency: Practice and Experience.*

[7] J. Demmel and K. Stanley. The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers. In David H. Bailey, Petter E. Bjørstad, John R. Gilbert, Michael V. Mascagni, Robert S. Schreiber, Horst D. Simon, Virginia J. Torczon y Layne T. Watson, editor, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 528–533. SIAM, 1995.

[8] C. G. J. Jacobi. Uber ein leichtes Verfahren die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch aufzulösen. *Journal für die Reine and Anbewante Mathematic*, 30:51–94, 1846.

[9] J. Demmel and K. Veselić. Jacobi's method is more accurate than QR. *SIAM J. Matrix Anal. Appl.*, 13:1204–1245, 1992.

[10] R. P. Brent and F. T. Luk. A systolic architecture for almost linear-time solution of the symmetric eigenvalue problem. Technical Report TR-CS-82-10, Department of Computer Science, Australian National University, Camberra, August 1982.

[11] G. H. Stewart. A Jacobi-like algorithm for computing the Schur decomposition of a nonhermitian matrix. *SIAM J. Sci. Stat. Comput.*, 4:853–864, 1985.

[12] *Paragon^{TM} System C Compiler User's Guide.* Intel Corporation, 1995.