Cache–Oriented Implementation of the Indefinite Block–Jacobi Method

Vjeran Hari[*1], **Sanja Singer**[**2], and **Saša Singer**[***1]

[1] Department of Mathematics, University of Zagreb, P.O. Box 335, 10002 Zagreb, Croatia.
[2] Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb, I. Lučića 5, 10000 Zagreb, Croatia.

Received 10 July 2006, revised 10 July 2006, accepted 15 July 2006

Key words indefinite Jacobi method, hyperbolic SVD, block algorithm, performance, speedup.
2000 Mathematics Subject Classification 65F15, 65Y20, 46C20, 15A18

One-sided indefinite Jacobi method for computing hyperbolic singular values of rectangular matrices can be used as an accurate eigensolver for Hermitian indefinite matrices, when combined with the Hermitian indefinite factorization.

This method is traditionally regarded as slow for serial computation. In recent times block-Jacobi methods have been proposed for parallel computation.

We propose several blocking strategies which are aimed to improve the performance of the method on a single processor with cache memory. Speedups of up to 40% have been achieved for large matrices.

1 Introduction

Classical Jacobi method for computing the eigendecomposition of a symmetric (or Hermitian) matrix H uses ordinary trigonometric plane rotations for systematic annihilation of offdiagonal elements, until the working matrix becomes numerically diagonal. If H is positive definite, this method computes the eigenvalues of H with high relative accuracy. But, if H is indefinite, the classical Jacobi method may be inaccurate.

An accurate modification is the so called indefinite (or hyperbolic) Jacobi method, proposed by Veselić [7], which observes the inertial structure of H.

Let H be of order m. The first phase of the algorithm computes the symmetric (or Hermitian) indefinite factorization of H

$$H = GJG^*, \quad J = \mathrm{diag}(j_{11}, \ldots, j_{nn}), \quad j_{ii} \in \{-1, 1\}, \quad i = 1, \ldots, n, \tag{1}$$

where G is an $m \times n$ matrix of full column rank, and J contains the nonsingular part of the inertia of H. This factorization follows easily from the Bunch–Parlett factorization of H (see [2, 3, 4]).

Note that nonzero eigenvalues of H are exactly all eigevalues of the pair $(A, J) := (G^*G, J)$.

In the second phase, algorithm simultaneously diagonalizes the matrix pair (A, J) by using elementary J-unitary rotations (either hyperbolic or trigonometric rotations). This diagonalization can be performed

- explicitly (two-sided algorithm on A), or
- implicitly (one-sided algorithm on G).

In practice one-sided algorithm is more accurate, and more than twice faster if vectorized routines are used (either compiler vectorization or BLAS 1 from Math Kernel Library).

* e-mail: hari@math.hr, Phone: +385 1 4605 748, Fax: +385 1 4680 335
** e-mail: ssinger@math.hr, Phone: +385 1 6168 215, Fax: +385 1 6156 940
*** Corresponding author: e-mail: singer@math.hr, Phone: +385 1 4605 745, Fax: +385 1 4680 335

2 One–Sided Indefinite Jacobi Algorithm

One-sided Jacobi algorithm is equivalent to the hyperbolic singular value decomposition (HSVD) of G,

$$G = U \left[\begin{array}{c} \Sigma \\ 0 \end{array} \right] V^*,$$

where U is unitary of order m,

$$\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n), \quad \sigma_1 \geq \sigma_2 \geq \cdots \sigma_n \geq 0,$$

is matrix of hyperbolic singular values, and finally, V is J-unitary matrix of order n (i.e., matrix which satisfies $V^*JV = J$).

If G is "well-behaved", i.e., if small relative changes of elements in G cause small relative changes in hyperbolic singular values σ_i, then one-sided indefinite Jacobi algorithm computes them with high relative accuracy.

On the other hand, the one-sided Jacobi algorithm is relatively slow compared to the fastest known diagonalization algorithms. But, in presence of two or more layers of memory with different speeds, many blocked algorithms show significant speedups. Ordinary matrix multiplication, which is a BLAS 3 operation, when performed as block multiplication, becomes several times faster then the elementwise multiplication. Similar reasoning motivates the construction of the indefinite block–Jacobi algorithm.

3 One–Sided Indefinite Block–Jacobi Algorithm

Block–Jacobi algorithms are usually considered in the context of parallel computing (see [6, 1]). The main objective of blocking for parallelization is data independancy. Our goal is quite the opposite: to **reuse** data which already resides in cache memory.

Suppose that G has the following block-column partition

$$G = [G_1, \ldots, G_p], \tag{2}$$

where the number of columns (block size) in G_i is n_i, and J diagonal matrix from (1), such that

$$J = \mathrm{diag}(J_1, \ldots, J_p),$$

and J_i is of order n_i. This column partition of G naturally induces a "square block" partition of $A = G^*G$.

In the spirit of [5], we propose two generalizations of the indefinite Jacobi algorithm

- block oriented algorithm,
- full block algorithm.

Both types of algorithms are performed in cycles, just like the ordinary non-blocked algorithm.

In each cycle of a block oriented algorithm we annihilate single elements of the working matrix, but the order of annihilation respects the block structure of A (for example, in block-cyclic order).

In full block algorithms we diagonalize pivotal submatrices

$$\widehat{A} = \left[\begin{array}{cc} A_{ii} & A_{ij} \\ A_{ij}^* & A_{jj} \end{array} \right]$$

of the working matrix A in some cyclic order.

4 Numerical Examples

We have run our tests on a computer with Pentium 4 660 processor (with 2 MB of cache memory) running under Windows XP Professional. We have used Intel FORTRAN compiler version 9.0.28, and BLAS and LAPACK routines contained in Intel Math Kernel Library 8.0.1. Our compiler optimization was set to aggressive optimization (`fast, optimize:5`).

Test matrices were randomly generated, by using LAPACK routine `DLARND` (uniformly distributed elements in $[-5, 5]$). The matrix order n varied from 500 to 4000. Block partition in (2) was equal-sized, with all n_i equal to the prescribed block size n_{cache}, except possibly for the last n_p which can be smaller than n_{cache}. Our block size n_{cache} varied from 8 to 128 columns in each block.

For $n \leq 500$ the non-blocked Jacobi algorithm was faster then any of the blocked algorithms, since big portions of the whole matrix reside in cache memory.

We also noticed that our "small", in-cache Jacobi algorithm, wich diagonalizes $\widehat{A}$ runs faster when we use compiler vectorization of loops instead of BLAS 1 routines. The gains are given in the following table.

Table 1 Full block algorithm: gain by vectorization over BLAS 1 for $n = 1000$ and block sizes 8–128.

Block size	8	16	24	32	40	48	56	64
Gain (%)	13.38	14.12	17.82	15.89	13.95	12.67	8.63	11.40
Block size	72	80	88	96	104	112	120	128
Gain (%)	9.69	7.68	7.44	10.35	6.80	6.18	2.87	5.80

We have carried out a number of tests with different versions of blocked algorithms. It should be noted that the speed of the algorithm is influenced by many implementation details, such as

- pivoting strategy in the symmetric indefinite factorization, (we used complete pivoting);
- partition of J, such that $J = I \oplus (-I)$;
- accumulation of J-unitary rotations for block transformations, etc.

At this point it is impossible to decide which one is the best, because several versions showed very similar speedups.

The following figures show typical speedups of block-oriented and full block algorithms.

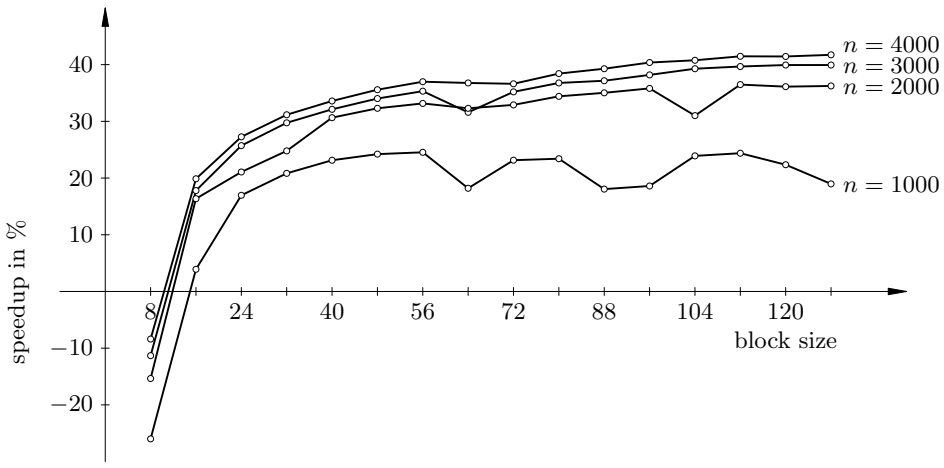

Fig. 1 Typical speedups for block-oriented algorithms.

5 Conclusion

Blocked algorithms are more then 40% faster then the non-blocked indefinite one-sided Jacobi algorithm for suitably large matrix dimensions. Our tests show that blocked algorithms also inherit relative accuracy from the non-blocked counterpart.

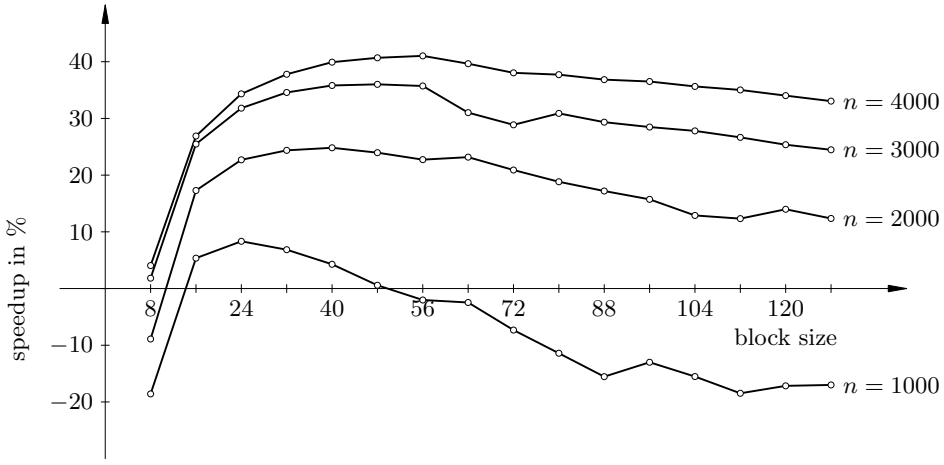

Fig. 2 Typical speedups for full block algorithms.

Acknowledgements This work was supported by grants 0037114 and 0037122 by Ministry of Science, Education and Sports, Croatia.

References

[1] P. Arbenz. I. Slapničar, An analysis of parallel implementations of the block-Jacobi algorithm for computing the SVD, in: D. Kalpić and V. Hljuz–Dobrić, eds., Proceedings of the 17th International Conference on Information Technology Interfaces, pp. 343–348, Pula, 1995.

[2] J. R. Bunch and L. C. Kaufman, Some stable methods for calculating inertia and solving symmetric linear systems, Math. Comp., **31** (1977), pp. 163–179.

[3] J. R. Bunch, L. C. Kaufman, and B. N. Parlett, Decomposition of a symmetric matrix, Numer. Math., **27** (1976), pp. 95–109.

[4] J. R. Bunch and B. N. Parlett, Direct methods for solving symmetric indefinite systems of linear equations, SIAM J. Numer. Anal., **8** (1971), pp. 639–655.

[5] V. Hari, Accelerating SVD block–Jacobi method, Computing, **75** (2005), pp. 27–53.

[6] G. Okša and M. Vajteršic, Efficient pre-processing in the parallel block–Jacobi SVD algorithm, Parallel Comput., **32** (2006), pp. 166–176.

[7] K. Veselić, A Jacobi eigenreduction algorithm for definite matrix pairs, Numer. Math., **64** (1993), pp. 241–269.