Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

A Parallel Jacobi-Davidson Method for Solving Generalized Eigenvalue Problems in Linear Magnetohydrodynamics

M. Nool, A. van der Ploeg

Modelling, Analysis and Simulation (MAS)

MAS-R9733 November 30, 1997

Report MAS-R9733
ISSN 1386-3703

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

A Parallel Jacobi-Davidson Method for Solving Generalized Eigenvalue Problems in Linear Magnetohydrodynamics

Margreet Nool and Auke van der Ploeg
<greta@cwi.nl>, <aukevdp@cwi.nl>

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

We study the solution of generalized eigenproblems generated by a model which is used for stability investigation of tokamak plasmas. The eigenvalue problems are of the form $Ax = \lambda Bx$, in which the complex matrices A and B are block tridiagonal, and B is Hermitian positive definite. The Jacobi-Davidson method appears to be an excellent method for parallel computation of a few selected eigenvalues, because the basic ingredients are matrix-vector products, vector updates and inner products. The method is based on solving projected eigenproblems of order typically less than 30.

The computation of an approximate solution of a large system of linear equations is usually the most expensive step in the algorithm. By using a suitable preconditioner, only a moderate number of steps of an inner iteration is required in order to retain fast convergence for the JD process. Several preconditioning techniques are discussed. It is shown, that for our application, a proper preconditioner is a complete block LU decomposition, which can be used for the computation of several eigenpairs. Reordering strategies based on a combination of block cyclic reduction and domain decomposition result in a well-parallelizable preconditioning technique. Results obtained on 64 processing elements of both a Cray T3D and a T3E will be shown.

1991 Mathematics Subject Classification: Primary: 65-04, 65F10, 65F15, 65F50, 65N25. Secondary: 65Y05, 65Y20.
1991 Computing Reviews Classification System: G.1.3
Keywords and Phrases: generalized eigenvalue problems, Krylov subspace methods, block tridiagonal systems, parallelization, preconditioning, restarting.
Note: Research carried out under project MAS2.3 – "Plasma physics simulation", and sponsored partly by the Cray Research Grants Program (project CRG 96.14) of NCF (Dutch National Computing Facilities Foundation) and by the Priority Program "Massaal Parallel Rekenen" (project 95MPR04) of NWO.

1. Introduction

Consider the generalized eigenvalue problem

$$Ax = \lambda Bx, \qquad A, B \in \mathcal{C}^{N_t \times N_t}, \tag{1.1}$$

where A and B are complex block tridiagonal N_t-by-N_t matrices and B is Hermitian positive definite. The number of diagonal blocks is denoted by N and the blocks are n-by-n, so $N_t = N \times n$.

Eigenvalue problems arise in many applications. We are particularly interested in generalized eigenvalue problems as they occur in magnetohydrodynamics (MHD). Such problems are generated by a finite-element spectral code called CASTOR. This code is applied intensively at the FOM Institute for Plasma Physics "Rijnhuizen" in Nieuwegein (near Utrecht) for the stability investigation of tokamak plasmas [9]. The physicists are particularly interested in accurate approximations of certain interior eigenvalues, called the *Alfvén spectrum* and their associated eigenvectors. Figure 1 shows the complete and the Alfvén spectrum of (1.1) for a small test problem with $N = 50$ and $n = 32$. In general, the subblocks of A are dense and N_t can be very large (realistic values are $N = 500$ and $n = 800$), so computer storage demands are very high. Till now, powerful shared memory machines

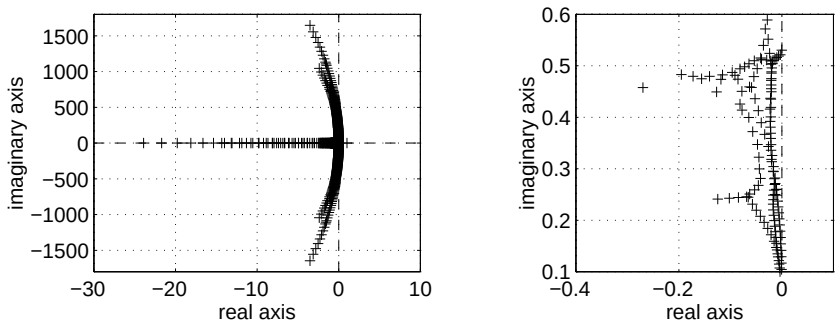

Figure 1: Entire(*left*) and Alfvén(*right*) spectrum of a small test problem: $N = 50$, $n = 32$.

with a huge amount of memory are used for this purpose. In this paper, we study an alternative possibility, namely, the feasibility of parallel computers with a large distributed memory for solving large generalized eigenvalue problems.

A promising method for computing selected eigenvalues of (1.1) is the Jacobi-Davidson (JD) algorithm [12, 2, 3, 4, 11]. The most time-consuming parts of this algorithm are matrix-vector and vector-vector operations, which are perfectly applicable to parallel execution. In Section 2 we briefly describe the JD algorithm.

At each step of the JD algorithm a system of linear equations, called the 'correction equation', has to be solved. Numerical experiments show that fast convergence to selected eigenvalues can be obtained by solving the correction equation to some *modest accuracy* only, by some steps of an inner iterative method, e.g. GMRES. For the MHD problems it is required to *precondition* the correction equation. This is discussed in Section 3. We have investigated a parallel preconditioner of Grote and Huckle [7], which is based on sparse approximate inverses. For our application, this method is not very successful because the inverse of $A - \sigma B$ (for some given fixed σ) cannot be approximated by a sparse matrix. We also have tried to exploit the sparsity of A and B by using the ILUT preconditioner of Saad [10], but this method is very hard to parallelize. A third approach, recently developed by one of us on the Cray T3D [13], is based on a parallelizable, direct method for solving block tridiagonal linear systems. In this approach, a block-reordering based on a combination of domain decomposition and cyclic reduction is combined with a complete LU decomposition (DDCR). This method appears to be very convenient for the kind of problems we are studying here.

In Section 4 it is shown that the availability of a complete LU decomposition gives us the opportunity to apply the Jacobi-Davidson method to a *standard* eigenvalue problem instead of a *generalized* eigenvalue problem. In this way, the eigenvalues that have to be computed form the dominant part of the spectrum, which makes them 'relatively easy'. Moreover, we can save a significant amount of memory.

As a basis for the development of the parallel code, we have taken a sequential `FORTRAN` code for the JD algorithm, which was developed by the late Albert Booten at CWI [2, 3, 4]. We started this project on the Cray T3D in Eagan, MN, USA; later on, we moved to the Cray T3E at HPαC, Delft. For comparison purposes, we used the Cray C90 at SARA, Amsterdam. The data structure for storing the matrices was modified in such a way that we could use optimized `BLAS` routines, as much as possible. Furthermore, the efficiency of reading the matrices A and B from file was improved considerably. This is described in Section 5.

In Section 6 we describe the parallel implementation of some important ingredients of the JD algorithm on the Cray T3D and T3E, and show some scaling results. In particular, we discuss the benefits of the *private data* versus *data sharing* programming styles on the T3D(E), and apply this

to the matrix-vector multiplication operation (the main operation in the JD algorithm) and to the preconditioner (DDCR) which we have chosen.

In Section 7 numerical results of the complete algorithm on 1 up to 64 processing elements of the Cray T3E are presented and analyzed. Comparisons are made with the performance of the sequential version on one CPU of the Cray C90. Conclusions are drawn in Section 8.

2. The Jacobi-Davidson Algorithm

For ease of presentation, we first describe briefly the Jacobi-Davidson algorithm for the standard eigenvalue problem in which $B = I$. This is followed by a description of the JD algorithm for the generalized eigenvalue problem. We assume that a target σ is given in the neighbourhood of which we want to find several eigenvalues with corresponding eigenvectors.

2.1 The standard eigenvalue problem

An eigenvector x is approximated by a linear combination of k search vectors v_j, $j = 1, 2, \cdots, k$, where k is very small compared with N_t. If the $N_t \times k$ matrix whose columns are given by v_j is denoted by V_k, the approximation to the eigenvector can be written as $V_k s$, for some k-vector s. The search directions v_j are made orthonormal to each other, hence $V_k^* V_k = I$.

Suppose that an approximation to an eigenvalue is denoted by θ. The vector s and the scalar θ are constructed in such a way that the residual vector $r = AV_k s - \theta V_k s$ is orthogonal to the k search directions. From this Rayleigh-Ritz requirement it follows that

$$V_k^* A V_k s = \theta V_k^* V_k s \Longleftrightarrow V_k^* A V_k s = \theta s.$$

In this way one obtains a 'projected' eigenvalue problem, in which the size of the matrix $V_k^* A V_k$ is k. By using a proper restart technique one makes sure that k stays so small that this problem can be solved by a dense method. The eigenvalue of the projected system that is closest to a preset value σ, is chosen as the approximate eigenvalue θ. In the context of our specific application, σ is chosen in the neighbourhood of the Alfvén spectrum.

At each step of the algorithm, a new search direction is constructed. Suppose that we have obtained an approximation $u = V_k s$ of the *true* eigenvector x associated with some eigenvalue λ. We assume that $\|u\| = 1$, hence $\theta = u^* A u$ is an approximation of λ. Let us define $P = uu^*$ being the orthogonal projector onto the subspace spanned by $\{u\}$. Then $I - P$ is the projector onto the orthogonal complement of span$\{u\}$, which is denoted by $u^\perp$. Any vector $x \in \mathcal{C}^n$ can be written as $x = x_1 + x_2$ with $x_1 \in$ span$\{u\}$ and $x_2 \in u^\perp$. We can scale x such that $x = u + z$ with $z \perp u$. In the JD algorithm a correction vector $z \in u^\perp$ is constructed. The restriction of A to $u^\perp$ is given by

$$A_{\mathrm{P}} = (I - P)A(I - P). \tag{2.1}$$

If we rewrite (2.1) and substitute the resulting expression for A into $Ax = \lambda x$, we obtain, using $Au - \theta u = r$, $z \perp u$, $Pu = u$ and $Pz = 0$:

$$(A_{\mathrm{P}} - \lambda I)z = -r + (\lambda - \theta - u^* A z)u. \tag{2.2}$$

Since also r is orthogonal to u, premultiplication of (2.2) with u^* yields $\lambda = \theta + u^* A z$. Note that λ is unknown and its best approximation will be θ. In this way, we obtain as the correction equation for z:

$$(I - P)(A - \theta I)(I - P)z = -r, \qquad u^* z = 0. \tag{2.3}$$

It is sufficient to solve (2.3) only approximately. This can be done by some steps of an iterative method, for example, GMRES. The speed of this iterative method can often be improved by using a preconditioning technique. When an approximate solution $\tilde{z}$ of (2.3) has been constructed, it is made orthogonal to the previous search directions, and the new search direction v_{k+1} is taken equal

to $\tilde{z}/\|\tilde{z}\|$. Then k is increased by one, and the new matrix $V_k^* A V_k$ is constructed by expanding the 'old' matrix by one new row and one new column.

As mentioned before, one has to make sure that the number of search directions k does not become too large. Therefore, if k has reached some value m, only the k_{min} most promising search directions are kept in memory. The values m and k_{min} are parameters which have to be chosen in advance.

2.2 The generalized eigenvalue problem

In [4, 11] a Jacobi-Davidson method for computing an eigensolution of (1.1) has been presented for $B \neq I$, where B is Hermitian positive definite. The approach is slightly different from the standard eigenvalue problem with $B = I$. Instead of looking for a correction $z \in u^{\perp}$, one looks for z in the B-orthogonal complement [1] of span$\{u\}$, denoted by $u^{\perp_B}$.

We assume the approximate eigenvector u to be normalized such that $||u||_B = 1$. The search directions are made B-orthonormal to each other, hence $V_k^* B V_k = I$. The residual vector $r = Au - \theta Bu$, with $u = V_k s$, is required to be orthogonal to the search directions, hence for the projected eigenvalue problem we obtain

$$V_k^* A V_k s = \theta s.$$

The new search direction is determined in a similar way as for the case $B = I$, leading to the correction equation

$$(A_{P_B} - \theta B)z = -r, \qquad u^* B z = 0 \tag{2.4}$$

in which $A_{P_B} = (I - P_B)A(I - P_B)$, where P_B is the B-orthogonal projector given by $P_B u = uu^* B$, and the residual vector r is given by $(A - \theta B)u$. Equation (2.4) can be rewritten as

$$(I - P_B)(A - \theta B)(I - P_B)z = -r, \qquad u^* B z = 0. \tag{2.5}$$

Algorithm 1 shows how the JD method for the solution of *several* eigenvalues can be implemented. Remarks on the algorithm:

- The algorithm is applied to the generalized eigenvalue problem $(A - \sigma B)x = \mu B x$, in which μ is defined as $\lambda - \sigma$. Since we want eigenvalues λ in the neighbourhood of σ, we are looking for μ with $|\mu|$ minimal. An approximation for μ is denoted by ν, and its corresponding eigenvector by u.

- The number of iteration steps that has been performed is denoted by it. The maximum allowed number of iterations is equal to $iter$.

- The value n_{ev} indicates the number of eigenvalues found so far that satisfy the acceptance criterion, and the parameter N_{ev} is the number of eigenvalues that we are looking for. The approximate eigenvalues ν that satisfy the acceptance criterion

  $$\|(A - (\sigma + \nu)B)u\|_2 \leq tol_{JD}.|\sigma + \nu|$$

  are referred to as μ_i, for $i = 1, \cdots, n_{ev}$. The algorithm stops when n_{ev} is equal to N_{ev}, or when it equals $iter$.

- In the actual implementation of the algorithm, precautions have to be taken in step 2: theoretically, it is possible that all eigensolutions of the projected system satisfy the acceptance criterion. If that would happen, in step 2 no new ν and corresponding approximate eigenvector u can be found. In that case, a vector $\tilde{z}$ is chosen that is not in the subspace spanned by V_k, and the algorithm proceeds at step 5.

[1] For a Hermitian positive definite matrix B the B-norm of a vector v is defined as $\|v\|_B = \sqrt{(v^* B v)}$. Two vectors v and w are said to be B-orthogonal if $w^* B v = 0$.

- In exact arithmetic, the vector r, which is computed in step 3 by $r := W_k^{A-\sigma B}s - \nu W_k^B s$, equals $(A-(\sigma+\nu)B)u$.
- The subspace spanned by V_k contains the eigenvectors corresponding to the eigenvalues found before, together with some search directions. In this way, implicit deflation is incorporated automatically. This deflation technique differs from the deflation technique described in [5] which uses explicit deflation. The latter technique can be more stable but requires more operations.
- Each time when the size k of the projected system has reached m, k is reduced to $k_{min}+n_{ev}$. The k_{min} search directions that are maintained during each restart, correspond with the k_{min} most promising eigenvalues of the projected system.
- We use Modified Gram-Schmidt [6] to orthonormalize vectors; 'call *MGSB* $[V_{k-1}, \tilde{z}]$' means that $\tilde{z}$ is made B-orthonormal to the columns of V_{k-1}. If the 2-norm of $\tilde{z}$ is much smaller after orthogonalization than before, one step of re-orthogonalization is used in order to reduce the effect of rounding errors. We use MGS instead of Classical Gram-Schmidt (CGS) although in the latter all inner products and vector updates can be computed independently, whereas this is not possible in MGS. However, MGS is more stable than CGS, and in our parallel implementation each *separate* inner product and vector update can be parallelized.
- As mentioned before, an existing code for the JD algorithm was taken as a basis for this project. As a consequence, the algorithm described here does not yet use Harmonic Ritz values [5, 12]. Harmonic Ritz values can give a better convergence behaviour, especially if restarts are used.

3. Preconditioners

A relative expensive part of the JD algorithm is the computation of an approximate solution of the correction equation (2.5). This equation is solved by an iterative method which requires a good preconditioner to achieve fast convergence. If M is some approximation of $A-\theta B$ then the projected matrix

$$M_p := (I-P_B)M(I-P_B)$$

can be considered as an approximation of $(I-P_B)(A-\theta B)(I-P_B)$. In [12] it is shown how, given the application of M^{-1}, the linear system $M_p y = d$ can be solved for some given d B-orthogonal to the Ritz vector u, such that y is B-orthogonal to u.

The matrix M should have the following properties:

1. It should be a proper approximation of $A-\theta B$, so that $M^{-1}(A-\theta B)$ resembles the identity matrix.
2. It should be cheap to compute, and its application, viz. the solution of the system $My = d$ for some given d, should not require significantly more operations than the computation of $(A-\theta B)d$.
3. It should not require a large amount of storage.
4. Both its construction and application should be parallelizable.

As has been stated in Section 2 we choose a target value σ in the neighbourhood of the Alfvén spectrum and we want to compute several eigenvalues in the neighbourhood of σ. The value θ in (2.5) is an approximation of a *true* eigenvalue and probably close to σ. That is why some approximation of $A-\sigma B$ in stead of $A-\theta B$ can be used as matrix M. This choice enables to compute a part of the Alfvén spectrum using the same M for several eigenvalues. The convergence behaviour indicates when a new target σ has to be chosen.

Algorithm 1. *Jacobi-Davidson for* $(A - \sigma B)x = \mu Bx$, B *Hermitian, positive definite. Parameters:* $iter, N_{ev}, tol_{JD}, k_{min}, m$ $(m \geq k_{min} + N_{ev})$, it_{GMRES}.

0: initialize
Choose an initial vector v_1 with $\|v_1\|_B = 1$; set $V_1 = [v_1]$;
$W_1^{A-\sigma B} = [(A - \sigma B)v_1]$; $W_1^B = [Bv_1]$; $k = 1$; $it = 1$; $n_{ev} = 0$.

1: update and solve projected system
Compute the last column and row of $H_k := V_k^* W_k^{A-\sigma B}$;
compute the eigenvalues $\nu_1, \cdots, \nu_k$ of H_k.

2: choose approximate eigensolution
Choose $\nu := \nu_j$ with $|\nu_j|$ minimal and $\nu_j \neq \mu_i$, for $i = 1, \cdots, n_{ev}$; compute a corresponding eigenvector s; scale s such that $\|V_k s\|_B = 1$; let u be the Ritz vector $V_k s$.

3: check accuracy
Compute the residual vector $r := W_k^{A-\sigma B} s - \nu W_k^B s$;
`if` $\|r\|_2 < tol_{JD}.|\nu + \sigma|$ `then`
  $n_{ev} := n_{ev} + 1$; $\mu_{n_{ev}} := \nu$; `if` $n_{ev} = N_{ev}$ `stop`;
  `goto` **2**
`else if` $it = iter$ `stop`
`end if`

4: solve correction equation approximately with it_{GMRES} steps of GMRES
Compute an approximate solution $\tilde{z}$ of
$(I - uu^*B)(A - (\sigma + \nu)B)(I - uu^*B)z = -r$ and $u^*Bz = 0$.

5: restart if projected system has reached its maximum order
`if` $k = m$ `then`
  Set $k = k_{min} + n_{ev}$. Construct $C \in \mathcal{C}^{m \times k}$ with as its columns eigenvectors of H_m which correspond to the n_{ev} eigenvalues accepted before and the k_{min} 'smallest' eigenvalues of H_m which have not been accepted. Orthonormalize columns of C; compute $V_k := V_m C$; $W_k^{A-\sigma B} := W_m^{A-\sigma B} C$; $W_k^B := W_m^B C$; $H_k := C^* H_m C$
`end if`

6: add new search direction
$k := k + 1$; $it := it + 1$; call *MGSB* $[V_{k-1}, \tilde{z}]$;
set $V_k = [V_{k-1}, \tilde{z}]$; $W_k^{A-\sigma B} = [W_{k-1}^{A-\sigma B}, (A - \sigma B)\tilde{z}]$; $W_k^B = [W_{k-1}^B, B\tilde{z}]$; `goto` **1**

In the next sections we discuss four different preconditioners. In realistic problems, the order of $A - \sigma B$ is very large, so it is too expensive to compute the inverse. The first approach tries to *approximate* the inverse by a sparse matrix. The matrices A and B in (1.1) are both block tridiagonal and therefore $A - \sigma B$ has the same block structure. The second preconditioning technique tries to exploit this structure by constructing an LU decomposition. Because σ is close to an eigenvalue, $A - \sigma B$ can be ill-conditioned. Therefore, in the third preconditioning technique, a complete block LU decomposition is combined with a pivoting strategy. The last preconditioning technique which will be discussed has better possibilities for parallelization.

3.1 SParse Approximate Inverse (SPAI)

In Grote and Huckle [7] a fully parallel algorithm for computing a sparse approximate inverse $\mathcal{M}$ of a matrix is described. If we use such a matrix as a preconditioner, then its application consists of a matrix-vector multiplication with $\mathcal{M}$, which is well-parallelizable. An approximate inverse $\mathcal{M}$ can be computed by solving a minimization problem of the form

$$\min \|(A - \sigma B)\mathcal{M} - I\| \tag{3.1}$$

for a given sparsity pattern of $\mathcal{M}$. If (3.1) is minimized in the Frobenius norm we obtain

$$\min \|(A-\sigma B)\mathcal{M}-I\|_F^2 = \min \sum_{k=1}^{N_t} \|(A-\sigma B)m_k - e_k\|_2^2, \tag{3.2}$$

where m_k and e_k are the k-th column of $\mathcal{M}$ and the identity matrix, respectively. The right-hand side sum is minimal if each term is minimal, which implies that all columns can be computed independently. The method starts with some prescribed sparsity pattern, e.g., $\mathcal{M} = I$ and at each step the number of nonzeros in the vector $m_k, k = 1, \cdots, N_t$ is extended. This is done by choosing a set of new promising indices which leads to a smaller l_2-norm of the residual $r_k^{col} = (A-\sigma B)m_k - e_k$, which requires the solution of a small least-squares problem. It is proved in [7] that there will always be such an index set which reduces $\|r_k^{col}\|_2$. Numerical experiments show that it is efficient to add a small number of nonzero entries to m_k per step. Following the suggestion in [7], we have chosen this number to be equal to 5. The process ends when $\|r_k^{col}\|_2 \leq$ a preset value ϵ, or when a prescribed number *pdrop* of nonzeros in a column is reached. For a detailed description of this algorithm we refer to [7]. The sparse approximate inverse of the matrix will not have a regular shape; it can have any sparsity pattern. Hence we cannot exploit the fact that $A-\sigma B$ is block tridiagonal.

We have done experiments with a small sparse block tridiagonal matrix $A-\sigma B$ of order 96: $N = 6$, $n = 16$. We used Booten's sequential code to find one eigenpair of this matrix. As an acceptance criterion for the Jacobi-Davidson method we demanded the residual norm of the eigenpair to be less than 10^{-8}. The target value we choose for this example is $\sigma = -0.35 + 0.6i$. The correction equation was solved approximately with 20 steps of preconditioned GMRES. Table 1 shows the number of iteration steps of the JD method required to find one eigenvalue for different *pdrop* and ϵ values. The last column shows the ratio of nonzeros in $\mathcal{M}$ and N_t^2. We see that a reasonable speed of converge

pdrop	ϵ	$\|(A-\sigma B)\mathcal{M}-I\|_F$	# JD steps	$nnz(\mathcal{M})/N_t^2$
32	.01	5.3	no convergence	0.26
48	.01	4.7	no convergence	0.36
64	.01	4.2	no convergence	0.44
96	.01	0.51	10	0.51
96	.001	0.46	11	0.59

Table 1: Results of SPAI preconditioning. $N = 6$, $n = 16$.

can only be reached when the approximate inverse matrix is almost dense. We observe that for the last two experiments with *pdrop* $= N_t$ some columns have reached the ϵ criterion before the maximum number per column was reached. We conclude that for our applications, the SPAI approach is not very successful because we do not succeed in approximating the inverse of $A-\sigma B$ by a sparse matrix.

3.2 ILUT

In [4], the ILUT preconditioner of Saad[10] is used. In this approach, a lower-triangular matrix L and an upper-triangular matrix U are constructed in such a way that $LU \approx A-\sigma B$. An approximation of the action of $(A-\sigma B)^{-1}$ consists of two triangular solves with the factors L and U. Since pivoting is not used, the matrix $L+U$ cannot have fill-in outside the block tridiagonal structure of $A-\sigma B$. In order to exploit the sparsity pattern within the separate blocks, only the entries in L and U that are not zero are stored. Hence operations with these factors require indirect addressing. The incomplete LU decomposition has two threshold parameters *pdrop* and τ. The parameter *pdrop* restricts the number of nonzero entries per row and per column in L and U; all elements smaller than τ are dropped during the incomplete factorization process.

In order to demonstrate this technique, we take a test problem from CASTOR that is used frequently in many reports [4, 11, 5, 13], in which $N = 26$ and $n = 16$. Again we use Booten's sequential code to

find one eigenpair of this matrix, and as an acceptance criterion we require the residual norm of the eigenpair to be less than 10^{-8}. Again the correction equation is solved with 20 steps of preconditioned GMRES. As target value we choose again $\sigma = -0.35 + 0.6i$. Table 2 shows the number of steps of the JD method for different *pdrop* and τ values. For our applications, more than 50% of the entries of the blocks on the diagonal and the sub- and super diagonal of $A - \sigma B$ are not zero. The last column shows the ratio of nonzeros of the factorization compared with the original matrix. If this ratio is larger than 1, the blocks in L and U cannot be called sparse. While first keeping τ fixed to the value

pdrop	τ	# JD steps	$nnz(L+U)/nnz(A-\sigma B)$
25	.0001	14	1.51
20	.0001	13	1.45
15	.0001	no convergence	1.25
25	.001	60	1.30
25	.01	no convergence	1.00

Table 2: Results of ILUT(*pdrop*,τ) preconditioning. $N = 26$, $n = 16$.

used in [4], *pdrop* is reduced as shown in Table 2. Next, *pdrop* is kept equal to 25, and we let τ grow. Choosing a larger value for *pdrop* is not very useful, because the LU decomposition is almost complete already. We may conclude that in our situation, fast convergence can only be obtained by choosing *pdrop* and τ such that the factorization is almost complete. Therefore, from now on we focus on a block LU decomposition of $A - \sigma B$, in which all elements of the blocks are stored, including those which are zero.

3.3 Standard LU approach

If a complete LU factorization is constructed, the total number of nonzero entries in $L + U$ is at most twice as high. In that case, the linear system with $A - \sigma B$ as coefficient matrix can be solved by two triangular solves only. However, in the JD algorithm we have to solve systems with $A - \theta B$ as coefficient matrix, in which θ is close to σ. The complete LU decomposition of $A - \sigma B$ can be used to precondition these linear systems. For a large number of diagonal blocks, the total number of multiplications required for the construction of L and U is approximately $\frac{7}{3}Nn^3$, and the number of multiplications required for performing the triangular solves with both L and U once is approximately $3Nn^2$. In the sequel of this paper, this approach is indicated as the standard LU approach.

The decomposition is performed on a block level. This enables us to use partial pivoting, which makes the preconditioner much more robust. In order not to disturb the block tridiagonal structure, the search for pivot elements is restricted to the blocks on the main diagonal. Another advantage is that indirect addressing is not required: both in the construction and application of the preconditioner we can use optimized `BLAS` routines. A drawback of the standard LU approach is that there are not many possibilities for parallelization.

3.4 The DDCR method

This approach has been introduced in [13]. For completeness and to introduce notation, we give a short description of this method. To improve parallelization possibilities, we use a reordering based on a combination of domain decomposition (DD) and cyclic reduction (CR). Let p be the number of processors that is actually used, and suppose that the integer $N_p = \lceil \frac{N}{p} \rceil$ [2] represents the number of diagonal blocks to be treated on each processor (except possibly the last processor, on which the number of diagonal blocks can be less). The sub-, super- and diagonal block of block row $(j-1)N_p + i$ are stored on the j-th processor, for $j = 1, \cdots, p$ and $i = 1, \cdots, N_p$. The first step of the DDCR method is to perform a block-reordering of both rows and columns based on a domain decomposition strategy with p non-overlapping subdomains.

[2] By $\lceil x \rceil$ we denote the smallest integer $\geq x$ and by $\lfloor x \rfloor$ the largest integer $\leq x$

First all diagonal blocks of $A - \sigma B$ are numbered, except the diagonal blocks $N_p + 1$, $2N_p + 1$, etc. Those blocks are numbered last. The matrix of the resulting system of linear equations can be partitioned as

$$\begin{bmatrix} A_1 & 0 & \cdots & 0 & A_{1,p+1} \\ 0 & A_2 & \cdots & 0 & A_{2,p+1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & A_p & A_{p,p+1} \\ A_{p+1,1} & A_{p+1,2} & \cdots & A_{p+1,p} & A_{p+1} \end{bmatrix}, \tag{3.3}$$

in which the matrices A_j are block tridiagonal with N_p diagonal blocks for $j = 1$ and with $N_p - 1$ diagonal blocks for $j = 2, \cdots, p-1$. The number of diagonal blocks in A_p is equal to $N-1-(p-1)N_p$. Note that A_{p+1} is block diagonal with $p-1$ blocks on the main diagonal. The corresponding system of equations is written as

$$\begin{bmatrix} C_1 & C_{12} \\ C_{21} & A_{p+1} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \tag{3.4}$$

where C_1 is a block diagonal matrix which has the block tridiagonal matrices A_j, $j = 1, \cdots, p$ on the main diagonal.

The second step of the DDCR method is to construct a block lower-triangular factor L and a block upper-triangular factor U in such a way that $A - \sigma B = LU$ and all blocks on the main diagonal of U are identity matrices. After copying the block lower- and upper-triangular part of $A - \sigma B$ to L and U respectively, $L + (U - I)$ has the block structure

$$L + (U - I) = \begin{bmatrix} L_1 & 0 & \cdots & 0 & U_{1,p+1} \\ 0 & L_2 & \cdots & 0 & U_{2,p+1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & L_p & U_{p,p+1} \\ L_{p+1,1} & L_{p+1,2} & \cdots & L_{p+1,p} & L_{p+1} + (U_{p+1} - I) \end{bmatrix}. \tag{3.5}$$

Fill-in blocks can only be generated in L_{p+1} and in U_{p+1}, and in $L_{p+1,j}$ and $U_{j,p+1}$ for $1 < j \leq p$. In [13] it is shown that after the construction of the LU decomposition, the matrix $L_{p+1} + (U_{p+1} - I)$ is block tridiagonal with $p-1$ main diagonal blocks, and the block sparsity pattern in $L_{p+1,j}$ and $U_{j,p+1}$ for $1 \leq j \leq p$ is regular. The number of extra fill-in blocks is approximately $(2p-2)/(3p)$ times the number of blocks in $A - \sigma B$. Such fill-in blocks are not present in the sequential standard LU approach: we have to pay this price for the possibilities for parallelization.

If we eliminate y_1 from (3.4), we obtain a system of linear equations

$$(A_{p+1} - C_{21}C_1^{-1}C_{12})y_2 = b_2 - C_{21}C_1^{-1}b_1 \tag{3.6}$$

in which the so-called Schur complement $A_{p+1} - C_{21}C_1^{-1}C_{12}$ is equal to $L_{p+1}U_{p+1}$. This block tridiagonal system is solved with block cyclic reduction. For completeness, and to introduce some notation used later, we give a short description of this approach. A more detailed description can be found in [8].

Suppose that the block tridiagonal linear system is given by

$$e_j x_{j-1} + d_j x_j + f_j x_{j+1} = b_j \qquad j = 1, \cdots, N, \tag{3.7}$$

in which e_1 and f_N are zero. Another block tridiagonal system, approximately twice as small, can be obtained by multiplying equation $2j-1$ by $-e_{2j}d_{2j-1}^{-1}$, equation $2j+1$ by $-f_{2j}d_{2j+1}^{-1}$ and adding the results to equation $2j$. The new system of equations can be represented by

$$-e_{2j}\tilde{e}_{2j-1}x_{2j-2} + d_{2j}^{new}x_{2j} - f_{2j}\tilde{f}_{2j+1}x_{2j+2} = b_{2j}^{new}, \qquad j = 1, \cdots, \lfloor \tfrac{N}{2} \rfloor.$$

Herein

$$\tilde{e}_{2j-1} = d_{2j-1}^{-1} e_{2j-1}, \tag{3.8}$$

$$\tilde{f}_{2j+1} = d_{2j+1}^{-1} f_{2j+1}, \tag{3.9}$$

$$d_{2j}^{new} = d_{2j} - e_{2j} d_{2j-1}^{-1} f_{2j-1} - f_{2j} d_{2j+1}^{-1} e_{2j+1}, \tag{3.10}$$

$$b_{2j}^{new} = b_{2j} - e_{2j} d_{2j-1}^{-1} b_{2j-1} - f_{2j} d_{2j+1}^{-1} b_{2j+1}. \tag{3.11}$$

This strategy can be repeated until only one block equation with one unknown vector of length n is left. Once x_{2j-2} and x_{2j} are computed, they may be substituted in equation $2j-1$ to compute x_{2j-1}.

To show which parts can be performed concurrently, we illustrate the preprocessing process in Table 3. Communication to left and right neighbouring processors is done concurrently, minimizing the number of barriers. At the beginning of the process all processors are involved, and at the end just one is busy. The cyclic reduction process is the last part of the block LU decomposition. If we would apply pure domain decomposition the complete last part is performed by just one processor.

For N_p large, the construction of a block LU decomposition of A_1 costs about $\frac{7}{3}n^3 N_p$ multiplications. In [13] it is shown that the rest of the block LU decomposition of (3.3) costs about $\frac{19}{3}n^3(N - N_p)$ multiplications. Hence the total number of multiplications required for the construction of L and U is approximately

$$(\tfrac{7}{3}N_p + \tfrac{19}{3}(N - N_p))n^3. \tag{3.12}$$

In the same way it can be shown that for large N_p the number of multiplications required for performing the triangular solves with both L and U once is approximately

$$(3N_p + 5(N - N_p))n^2. \tag{3.13}$$

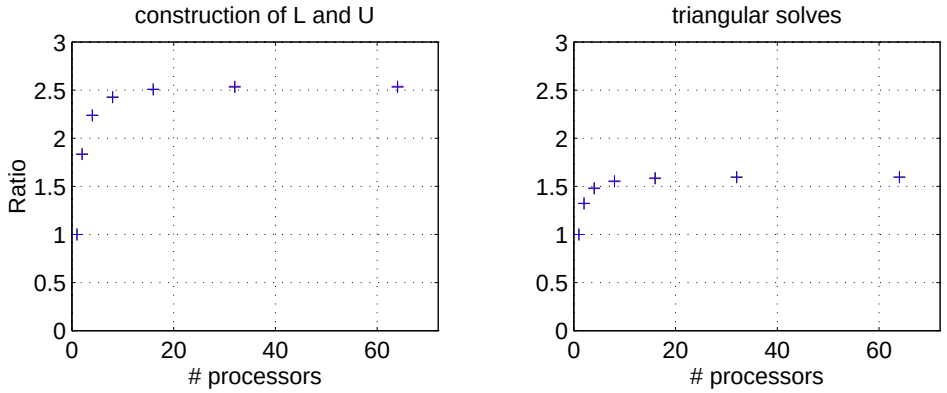

Figure 2: The ratio of number of flops required for the DDCR method and for the standard LU approach.

In Appendix I, Tables 12 and 13 give a review of the number of block matrix operations required for preprocessing and the solution process as function of N_p and p. In Figure 2 the increase of the number of floating point operations of the parallel DDCR method compared with the sequential standard LU approach is shown, both for the construction of L and U (denoted by DDCR), which varies from 1.8 up to 2.6, as well as for the triangular solves (referred to as SOLDDCR), going from 1.3 up to 1.6. The extra work required to deal with the fill-in blocks has to be compensated by using several processors.

p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7
$d = LU$ $f = d^{-1}f$		$d = LU$ $e = d^{-1}e$ $f = d^{-1}f$		$d = LU$ $e = d^{-1}e$ $f = d^{-1}f$		$d = LU$ $e = d^{-1}e$ $f = d^{-1}f$	
Send e to the left $(e(p_i) = e^R(p_{i-1}))$ and send f to the right $(f(p_i) = f^L(p_{i+1}))$. `BARRIER`							
$e_0 := e^R$	$e_0 := e^R$ $f_0 := f^L$ $d = d - f.e^R$ $d = d - e.f^L$	$e_0 := e^R$ $f_0 := f^L$ $x = -e^R.e$ $y = -f^L.f$	$e_0 := e^R$ $f_0 := f^L$ $d = d - f.e^R$ $d = d - e.f^L$	$e_0 := e^R$ $f_0 := f^L$ $x = -e^R.e$ $y = -f^L.f$	$e_0 := e^R$ $f_0 := f^L$ $d = d - f.e^R$ $d = d - e.f^L$	$e_0 := e^R$ $f_0 := f^L$ $x = -e^R.e$ $y = -f^L.f$	$f_0 := f^L$ $d = d - e.f^L$
Send x to the right $(x(p_i) = x^L(p_{i+1}))$ and send y to the left $(y(p_i) = y^R(p_{i-1}))$. `BARRIER`							
	$f := y^R$		$e := x^L$ $f := y^R$		$e := x^L$ $f := y^R$		$e := x^L$
End of the first step of the cyclic reduction process.							
	$d = LU$ $f = d^{-1}f$				$d = LU$ $e = d^{-1}e$ $f = d^{-1}f$		
Send e over 2 PEs to the left $(e(p_i) = e^R(p_{i-2}))$ and send f over 2 PEs to the right $(f(p_i) = f^L(p_{i+2}))$. `BARRIER`							
	$e_0 := e^R$		$e_0 := e^R$ $f_0 := f^L$ $d = d - f.e^R$ $d = d - e.f^L$		$e_0 := e^R$ $f_0 := f^L$ $x = -e^R.e$ $y = -f^L.f$		$f_0 := f^L$ $d = d - e.f^L$
Send x over 2 PEs to the right $(x(p_i) = x^L(p_{i+2}))$ and send y over 2 PEs to the left $(y(p_i) = y^R(p_{i-2}))$. `BARRIER`							
			$f := y^R$				$e := x^L$
End of the second step of the cyclic reduction process.							
			$d = LU$ $f = d^{-1}f$				
Send e_0 over 4 PEs to the left $(e(p_i) = e^R(p_{i-4}))$ and send f_0 over 4 PEs to the right $(f(p_i) = f^L(p_{i+4}))$. `BARRIER`							
			$e_0 := e^R$				$f_0 := f^L$ $d = d - e.f^L$
End of the third step of the cyclic reduction process.							
							$d = LU$

Table 3: The construction of L and U for p=N=8, which corresponds to block cyclic reduction. On the j-th processor, d, e and f correspond to d_j, e_j and f_j in (3.7), respectively. e_0 and f_0 are extra fill-in blocks which correspond to $\tilde{e}_{2j-1}$ and $\tilde{f}_{2j+1}$ in (3.8) and (3.9). x and y are auxiliary arrays used for data transfer. The superscripts ^{L} and ^{R} denote whether the matrix is received from a left or right processor, respectively.

4. Going from a generalized eigenvalue problem to a standard eigenvalue problem

Writing (1.1) as $(A - \sigma B)x = (\lambda - \sigma)Bx$ we have

$$x = (\lambda - \sigma)(A - \sigma B)^{-1}Bx.$$

We construct a complete LU decomposition of $A - \sigma B$. Hence if rounding errors in this decomposition can be neglected, $(A - \sigma B)^{-1} = (LU)^{-1}$. The matrix-vector multiplication with $(A - \sigma B)^{-1}$ then consists of two triangular solves and we can replace the generalized eigenvalue problem by a standard eigenvalue problem. If we define Q as $(LU)^{-1}B$ we obtain the standard eigenvalue problem

$$Qx = \mu x, \qquad \text{with } \mu = \frac{1}{\lambda - \sigma} \Leftrightarrow \lambda = \sigma + \frac{1}{\mu}. \tag{4.1}$$

By applying the JD algorithm to (4.1) instead of to the generalized problem, we can save a significant amount of memory. Moreover, the eigenvalues we are interested in form the dominant part of the spectrum, which makes them 'relatively easy'.

Algorithm 2 shows how the JD method for the solution of several eigenvalues can be implemented. The parameters have a similar meaning as in Algorithm 1. The difference with Algorithm 1 is that the matrix W_k^B does not have to be stored, and the 2-norm instead of the B-norm is used. MGS is an abbreviation for the orthogonalization by Modified Gram-Schmidt [6]. '*Call* $MGS[V_{k-1}, \tilde{z}]$' means that $\tilde{z}$ is made orthonormal to the columns of V_{k-1}. Note that the matrix-vector multiplication with Q consists of the matrix-vector multiplication with B combined with two triangular solves with L and U.

4.1 What if $(LU)^{-1}$ equals $(A - \sigma B)^{-1}$ only approximately?

In practice, one has to be careful because small pivot elements can be generated during the decomposition of $A - \sigma B$, especially when σ is very close to an eigenvalue. In that case, taking $Q = (LU)^{-1}B$ in (4.1) may influence the computed spectrum, and we therefore have to use a more accurate matrix-vector multiplication with Q. Suppose that for some vector d the computed solution of

$$(A - \sigma B)x = d \tag{4.2}$$

is not accurate enough. One possible remedy is to perform one step of iterative refinement in order to obtain a correction vector. If this does not give a solution that is accurate enough, one can solve (4.2) by using some steps of GMRES which uses the LU decomposition as a preconditioner. However, if several steps of GMRES are required, it is probably more efficient not to reformulate $Ax = \lambda Bx$ as a standard eigenvalue problem. In that case, it is better to apply Algorithm 1 to the generalized eigenvalue problem.

Note that in the inner iteration to solve the correction equation, the accurate matrix-vector multiplication with Q is not needed. Since an *approximate* solution of the correction equation is sufficient, within the inner iteration the cheaper matrix-vector multiplication with $(LU)^{-1}B$ can be used.

5. Data Organization

In Section 5.1 we discuss which storage schemes are to be preferred for CASTOR's MHD matrices A and B in (1.1). For large problems with a lot of data, the I/O is often very expensive. Therefore, it is important to read data files efficiently. In Section 5.2 we describe how this can be organized such that data files become independent of the number of processors that will be used. For information about the distribution of arrays over the processors and the way communication is performed, we refer to Section 6.1.

5.1 Data Format: CRS format versus dense block tridiagonal format

The Compressed Row Storage (CRS) format puts the subsequent nonzeros of the matrix rows in contiguous memory locations. To describe a sparse matrix A, we need 3 vectors: one for complex

Algorithm 2: *Jacobi-Davidson for* $Qx = \mu x$.
Parameters: $iter, N_{ev}, tol_{JD}, k_{min}, m$ $(m \geq k_{min} + N_{ev})$, it_{GMRES}.
0: initialize
Choose an initial vector v_1 with $\|v_1\|_2 = 1$; set $V_1 = [v_1]$;
$W_1 = [Qv_1]$; $k = 1$; $it = 1$; $n_{ev} = 0$.
1: update and solve projected system
Compute the last column and row of $H_k := V_k^* W_k$;
compute the eigenvalues $\theta_1, \cdots, \theta_k$ of H_k.
2: choose approximate eigensolution
Choose $\theta := \theta_j$ with $|\theta_j|$ maximal and $\theta_j \neq \mu_i$, for $i = 1, \cdots, n_{ev}$; compute a corresponding eigenvector s with $\|s\| = 1$; let u be the Ritz vector $V_k s$.
3: check accuracy
Compute the residual vector $r := W_k s - \theta u$;
`if` $\|r\|_2 < tol_{JD}.|\theta|$ `then`
$n_{ev} := n_{ev} + 1$; $\mu_{n_{ev}} := \theta$; `if` $n_{ev} = N_{ev}$ `stop`;
`goto` **2**
`else if` $it = iter$ `stop`
`end if`
4: solve correction equation approximately with it_{GMRES} **steps of GMRES**
Compute an approximate solution $\tilde{z}$ of
$(I - uu^*)(Q - \theta I)(I - uu^*)z = -r$ and $u^* z = 0$.
5: restart if projected system has reached its maximum order
`if` $k = m$ `then`
Set $k = k_{min} + n_{ev}$. Construct $C \in \mathcal{C}^{m \times k}$ with as its columns eigenvectors of H_m which correspond to the n_{ev} eigenvalues accepted before and the k_{min} 'largest' eigenvalues of H_m which have not been accepted. Orthonormalize columns of C; compute $V_k := V_m C$; $W_k := W_m C$; $H_k := C^* H_m C$
`end if`
6: add new search direction
$k := k + 1$; $it := it + 1$; call $MGS\,[V_{k-1}, \tilde{z}]$;
set $V_k = [V_{k-1}, \tilde{z}]$; $W_k = [W_{k-1}, Q\tilde{z}]$; `goto` **1**

floating point numbers (*mat_A*), and two for integers (*colind_A*, *rowptr_A*). The *mat_A* vector stores the values of the nonzero elements of matrix A, as they are traversed in a row-wise fashion. The *colind_A* vector stores the column indices of the elements in *mat_A*. The *rowptr_A* vector stores the locations in the *mat_A* vector that start a row. For more details and examples, we refer to [1].

CRS format has the advantage that it can be used to store general sparse matrices. However, the DDCR preconditioner is completely based on the block tridiagonal form of $A - \sigma B$. This implies that, for our applications, CRS format can only be advantageous if the blocks of A and B are sparse enough. Table 4 shows the execution times for different matrix-vector multiplications on a single processor of both a Cray C90 and a Cray T3D for matrices A and B generated by the CASTOR code. These results indicate that these platforms require different approaches. For the Cray C90 the matrix-vector multiplication applied to B (sparsity of the sub-, super-, and diagonal blocks $\approx 20\%$) stored in CRS format (CRS(B)) takes more time than those in which all entries of these blocks are considered nonzero (`DENSE`). A third approach has been timed on the Cray C90, i.e., the Jagged Diagonal Storage (JDS) format [1]. It is well-known that for true sparse matrices this storage format is efficient on vector machines and the results of Table 4 show that for small block sizes this is the fastest solution.

On the Cray T3E, which is not a vector computer, CRS format is to be preferred for both A and B. We note that when Algorithm 2 is applied without the more precise matrix-vector multiplication

Execution times in μ-seconds on a Cray C90, $N = 40$.

MATVEC	$n = 16$	$n = 32$	$n = 48$	$n = 64$	$n = 80$
DENSE	638	1423	2777	4675	7074
CRS(B)	1547	3315	5263	7385	10208
JDS(B)	274	1493	2573	6456	7425
Execution times in μ-seconds on a Cray T3E, $N = 40$.					
MATVEC	$n = 16$	$n = 32$	$n = 48$	$n = 64$	$n = 80$
DENSE	4490	14577	32055	54666	93924
CRS(B)	1421	4979	10806	15814	24374
CRS(A)	2469	9874	22518	33733	52861
Number of nonzeros in A and B, N=40.					
	$n = 16$	$n = 32$	$n = 48$	$n = 64$	$n = 80$
nnz(B)	5934	22778	50532	89196	138770
nnz(A)	11392	47906	109542	196300	308180

Table 4: Times for the matrix-vector multiplication on a single processor of both a Cray C90 and a Cray T3E. Results for the block tridiagonal form (dense blocks) are denoted by DENSE. Results for CRS format are denoted by CRS(A) and CRS(B) for A and B respectively. Results for B stored in JDS format are denoted by JDS(B). The integers nnz(A) and nnz(B) denote the number of nonzeros.

with Q (see Section 4.1), the matrix A only occurs in the LU factorization of $A - \sigma B$. This implies that A can be overwritten by $A - \sigma B$, and next by its LU decomposition. Since the blocks in the decomposition are not sparse, dense block tridiagonal storage is used for A.

Otherwise, when the more accurate Q is used, the multiplication with $A - \sigma B$ is a part of the multiplication with Q. CRS format is to be preferred on the Cray T3E, although the gain is less obvious than for the more sparse matrix B.

5.2 Reading data from files

Our FORTRAN implementation reads the data files

rowptr_A, *colind_A*, *mat_A*, *rowptr_B*, *colind_B* and *mat_B*,

which must contain unformatted CRS data for both matrices A and B. Unformatted READ appears to be much faster than formatted READ. Moreover, unformatted files need less disk storage than formatted files. Data for a single block row(i.e., three blocks of dimension n) must appear row-wise in a single record. This enables to keep data files independent of the number of processors the program should run on. Currently, the matrices are read from file sequentially (in the near future we hope to use parallel I/O): the first PE reads its part(s) from file, then the second PE reads its corresponding part and so on. This approach enables to distribute data communication free. We remark that it is not necessary that the nonzero entries are equally distributed along the block rows and, as a consequence, along the processors. Therefore, an upper bound must be given for the maximum number of nonzeros per processor. In our experiments we took the number of nonzeros of a matrix divided by the number of processors, multiplied by a factor 1.2.

In our experiments, a well-chosen target can deliver 10 up to 20 eigenvalues. Of course, this number strongly depends on the parameters k_{min} and m in Algorithm 1 and 2. For a restart with another target σ', there are two possibilities:

- Make a fresh start with a new target. Reread data from files.
- Calculate $A - \sigma' B = (A - \sigma B) - (\sigma' - \sigma)B$, if $A - \sigma B$ is available.

As described in Section 5.1, the matrix A may sometimes be overwritten by the factors L and U. A choice between the first and second possibility, is then a choice between execution time versus memory

usage. Since reading the files takes just a few percent of the total execution time, and the amount of memory still appears to be the largest bottleneck (cf. Section 7) a fresh restart will often be the best choice.

6. The parallel implementation

We started this project with the parallelization of the original code [2, 3, 4] on a Cray T3D in Eagan, MN, USA. The last part of the development has been done on a Cray T3E in Delft. An important reason for switching form Eagan to Delft, was the response time of the Cray T3D, which was very slow, especially in the afternoon.

There are some important differences between both machines, which are listed in Table 5. On the

	Cray T3D	Cray T3E
# processors	128	80 (72 in parallel)
Clock period	6.67 ns clock	3.33 ns clock
Peak Performance / processor	150 Mflop/s	600 Mflop/s
Total Peak Performance	19.2 Gflop/s	33.6 Gflop/s
Local Memory / processor	64 Mbytes	128 Mbytes
Compiling System	Cray CF77	Cray CF90
CRAFT (for `SHARED` programming)	supported	not supported
GiGaRing channels	not available	available
Situated at	Eagan	HPαC, Delft

Table 5: Main differences between Cray T3D and T3E

Cray T3D we used a programming technique involving so-called CRAFT directives, which are not yet supported on the Cray T3E. The moment we switched from T3D to T3E, we already had decided to continue with the `MESSAGE PASSING` approach, because that appeared to be more efficient for our application, as is pointed out in Section 6.1. In the sequel, we show results from both machines, but a systematic comparison has not been made.

6.1 Data distribution and communication: Data Sharing versus Private Data

One can roughly distinguish two different programming styles on the Cray T3D(E). The first one, based on data sharing, is comparable with the virtual shared memory model `HPF` (High Performance Fortran), the second one is closely related to `MESSAGE PASSING` (cf. `PVM` and `MPI`). Both methods have been well-tuned for the Cray T3D(E) and are expected to be faster than using `HPF`, `PVM` or `MPI`.

The programming styles differ in the distribution of arrays:

- DATA SHARING and CRAFT directives:

  The Cray T3D(E) can be considered as a distributed shared memory machine. By this we mean that all arrays can be distributed over the processors in a user's prescribed way. Every processor has access to all array elements even when they are stored on other PEs. Below, we give a simple example of a `SAXPY` operation on `SHARED` data. In this example, the array length `N` should be a multiple of the number of processors.

```
      PARAMETER     ( N = 512, ALPHA = 2.0 )
      REAL            X(N), Y(N)
CDIR$ GEOMETRY G( :BLOCK )
CDIR$ SHARED (G):: X, Y
*
CDIR$ DOSHARED(I) ON Y(I)
      DO I = INX, N
         Y(I) = Y(I) + ALPHA * X(I-INX+1)
      END DO
```

The arrays X and Y are equally distributed. If this program is executed on p =N$PES processors (N$PES is a special run time constant to find out how many processors are running the program), then due to geometry G, the first processor will have the first N/p elements of the arrays X and Y, the second processor the next N/p elements and so on. In case $p > 1$ & 1 < INX < N/p, communication is needed implicitly. The compiler calculates where desired elements can be found, but that causes a certain delay. On each processor, except the first one, the DO loop requires INX-1 elements of X which are present on a previous (neighboring) processor (located in the last INX-1 positions of X over there). We speak about remote data, when data is not present on the same processor on which the computations are performed. Operating on remote data is much slower than on local data, because processors have to communicate. Communication is invisible, except that the performance becomes much lower. We remark that in the above example, a calculation is performed on the processor which stores element Y(I), I=INX,N. In the sequel, we use the term SHARED if we refer to this programming technique. An advantage of this technique is that it leads to portable code: work and data are distributed over the processors by using directives. On other machines, for example on work stations, these directives are just comment lines. Hence the code that parallelizes on the CRAY T3D runs without any modification on other machines.

We have improved the performance of our code by copying remote data into an auxiliary array, which is local to the processor. Then we can use BLAS routines which increase the computational speed.

The dimensions of SHARED arrays have to be declared as a power of two. The matrices A and B have a block tridiagonal structure on which the DDCR preconditioner is based. Therefore, it is desirable to consider a block as the smallest computational unit. However, typical block sizes for MHD problems are $n_f \times 16$, where the number of Fourier modes n_f may not be a power of two. The block dimension should then be extended to a power of two. Moreover, both the number of diagonal blocks and the number of PEs must be a power of two. This can lead to large unwanted memory extensions, especially for multi-dimensional arrays.

- MESSAGE PASSING and PRIVATE DATA:

  In a second approach, dealing with PRIVATE or local data, the user takes care of the communication: all data is stored locally on a processor and arrays are called PRIVATE. All communication between PEs must be done explicitly by calling communication routines.

  Below, we give a MESSAGE PASSING implementation of the SAXPY operation above, using the transfer routine SHMEM_PUT.

```
      INTEGER         I, INX, MY_PE, NL
      PARAMETER     ( N = 512, NL = N/N$PES, ALPHA = 2.0 )
      REAL            X(NL), Y(NL), AUX(INX-1)
      INTRINSIC       MY_PE
*
      COMMON / COM / X, Y, AUX
*
      IF( MY_PE().LT.N$PES-1 )THEN
*
* The last INX-1 elements of X are sent to the next processor and
* copied into array AUX.
*
         IPUT = SHMEM_PUT( AUX, X(NL-INX+2), INX-1, MY_PE()+1)
*
      END IF
*
* Next, those elements of Y are updated which need no communication.
```

```
* Data transfer and computation of Y can (partly) be done simultaneously.
*
      DO I = INX, NL
         Y(I) = ALPHA * X(I-INX+1) + Y(I)
      END DO
*
* A barrier is needed to be sure that all data sent by SHMEM_PUT has arrived.
*
      CALL BARRIER()
*
      IF( MY_PE().GT.0 )THEN
*
* The first INX-1 elements of Y can now be updated without communication
*
         DO I = 1, INX-1
            Y(I) = ALPHA * AUX(I) + Y(I)
         END DO
      END IF
```

In this example, data is private to the processor and communication routines like SHMEM_GET and SHMEM_PUT can be used to send data from one processor to another. The local arrays X and Y are both of dimension NL, which is equal to the original N divided by p. The commonblock COM is used to prescribe the order in which the arrays X, Y and AUX are stored; in this way, the bases addresses of the arrays are the same on each PE. This is necessary if processors write in other processors' memories. Again, if INX > 1 & $p > 1$ communication is required, but in this MESSAGE PASSING case it must be done explicitly. The use of the transferring routines SHMEM_GET and SHMEM_PUT is slightly different: SHMEM_GET is called by the receiving processor and it returns when all desired data has been transferred. SHMEM_PUT is called by the sending processor. It can return before data has arrived on the receiving PE. It appears that by changing the order of operations, e.g., by sending data at the moment it has been calculated rather than to wait until it is needed by other processors, idle time can be reduced, because in that case data transfer is approximately for free. However, one must take care to operate on the right data. Barriers can be used to synchronize the process: when a barrier is reached by some processor, it has to wait until the other processors involved in that barrier have arrived and all data transfer has been completed. A surplus of barriers will slow down the computational process.

By using PRIVATE data instead of SHARED data we can avoid large unwanted memory extensions. From the timing results for the matrix-vector multiplications presented in Section 6.2 it will be clear that not only for matrices, but also for vectors PRIVATE data is to be preferred.

In Figure 3, we show for different block sizes the communication times (in micro seconds). For both the SHMEM_GET and the SHARED copy, two complex vectors, each of length n, have to be transferred from one processor to another. The communication time does not increase when p becomes larger than 4, because only nearest neighbor communication is needed. We observe that for these short vectors, communication on private data is much faster than on shared data, especially for larger block sizes. These results correspond with experiences of Van der Steen [14], who shows that many virtual shared memory models are less efficient than their MESSAGE PASSING counter parts.

6.2 Matrix-vector multiplication

For the matrix-vector multiplication we have made four different implementations for the Cray T3D:

- For the matrices, both dense block tridiagonal format and CRS format are used. They will always be PRIVATE: each PE has its own part. In Section 5.1 storage formats have been discussed.
- The vectors can be both SHARED and PRIVATE.

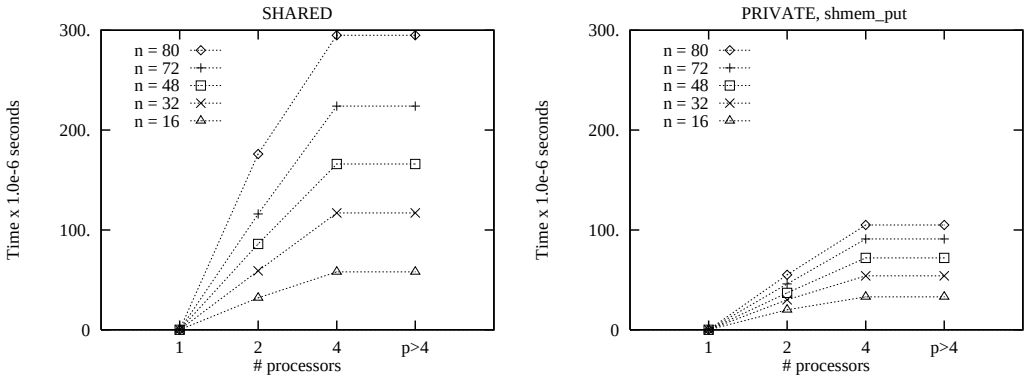

Figure 3: Comparison of communication times on the Cray T3D for `SHARED` data versus `PRIVATE` data for the complex matrix-vector multiplication with a block tridiagonal matrix of order n. The communication times are independent of the number of diagonal blocks per PE.

For the dense block matrix-vector multiplication, we use for each block row the efficient `BLAS` routine `ZGEMV`. In Figure 4 Mflop-rates for the four different implementations described above are given. The highest values are achieved when $n = 80$, associated with the largest number of floating point operations (flops).

The matrix used in the CRS matrix-vector multiplication of Figure 4 has a block tridiagonal structure, where each block contains only entries on its main diagonal. This matrix has been constructed only for testing purposes. In this project we started with MHD problems of which the blocks of matrix B of (1.1) contain about 5% nonzeros. For this matrix in CRS format much lower performance results are obtained due to indirect addressing and the small number of nonzeros per row (only 3) compared with the dense block implementation.

The influence of communication in combination with data distribution becomes most visible for the matrices stored in CRS format. However, also in this case, communication is minimal, since there are no nonzero entries outside the tridiagonal blocks, and therefore, on each processor only a small part of the vector must be available. Clearly, the number of flops is small compared with the amount of data transfer. We also observe that going from 32 to 64 PEs the performance growth is less impressive than going from 16 to 32 PEs due to data transfer. For the dense case this effect is much smaller, because the time needed for flops dominates the communication time.

From Table 6, we may also conclude that the matrix-vector multiplication on the Cray T3D is perfectly scalable. Here, the number of diagonal blocks is always twice the number of processors. So, if the problem size *and* the number of PEs are doubled, we may expect the execution time to remain unchanged. Again the same test matrix as described above is used.

	Dense Matrix Storage Shared Vector			Dense Matrix Storage Local Vector			CRS Matrix storage Shared Vector			CRS Matrix storage Local Vector		
p	16	32	64	16	32	64	16	32	64	16	32	64
n												
16	424	415	422	272	274	273	181	174	176	113	110	116
32	1136	1136	1154	801	803	799	284	282	282	186	185	199
48	3797	3675	3691	2847	2845	2845	465	465	465	331	327	353

Table 6: Execution times in μ-seconds on a Cray T3D for different complex matrix-vector implementations. $N = 2p$. For the CRS format the blocks have 3 nonzero entries per row, for the dense matrix-vector product this number is $3 \times n$.

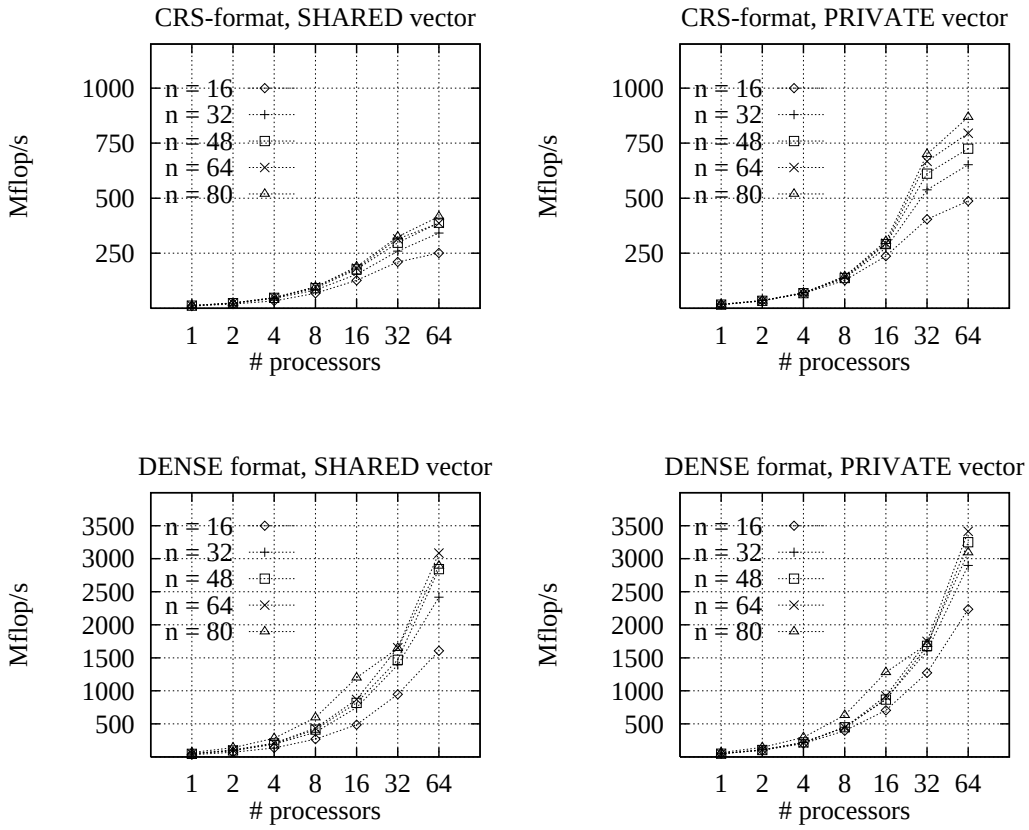

Figure 4: Mflop-rates for four different implementations of a matrix-vector multiplication with block tridiagonal matrices on the Cray T3D. Only the main diagonal elements of the sub-, super- and diagonal blocks are not zero. N=64.

6.3 Timing results of the preprocessing routine DDCR and the solution process routine SOLDDCR
To demonstrate the parallel performance of the DDCR method, Figure 5 shows the wall clock time in seconds on a Cray T3D, both for the construction (denoted by DDCR) and the application of L and U (denoted by SOLDDCR). We used CRAFT directives for the distribution of work and data over the processors (shared data). The number of processors p increases from 2 to 64, and the number of diagonal blocks N is equal to $16p$. Hence for a perfectly scalable algorithm, the time would be constant.

From the results it appears that the wall clock time does not increase strongly with the number of processors, hence the parallel performance is quite good. By using more processors, we can increase the dimension of the problem with a factor 32, whereas the wall clock time increases by a factor less than 2. If 64 processors are used, the Mflop-rate for the construction of L and U is approximately 3600.

We have tested three different implementations. The first one is the data sharing code. As we have argued in Section 6.1, we prefer to use private data, in which case we have to transfer data explicitly from one PE to another. The second and third implementation both using `MESSAGE PASSING`, differ in the way data is transferred: one calls `SHMEM_GET`, the other calls `SHMEM_PUT`. For more details on these routines, we refer to Section 6.1.

Figure 6, giving the speed-ups of three different implementations of the DDCR method compared with the fastest standard LU approach on a single processor, shows that the `SHMEM_PUT` version is to

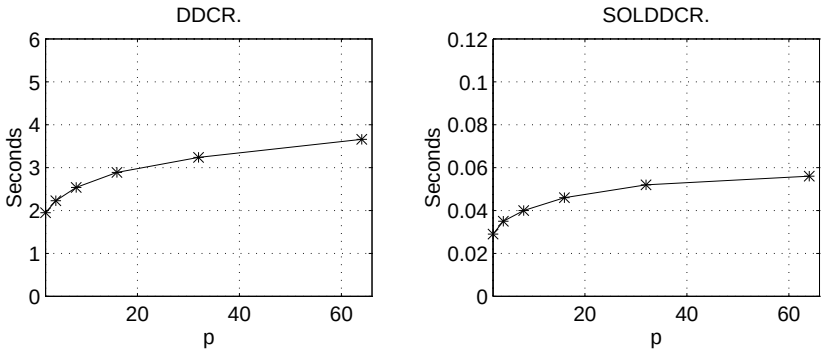

Figure 5: Wall clock times in seconds on the Cray T3D required for construction (left) and application (right) of the DDCR preconditioner, $n = 64$, $N = 16p$, $p = 2, 4, 8, 16, 32$ and 64. Results of the data sharing implementation.

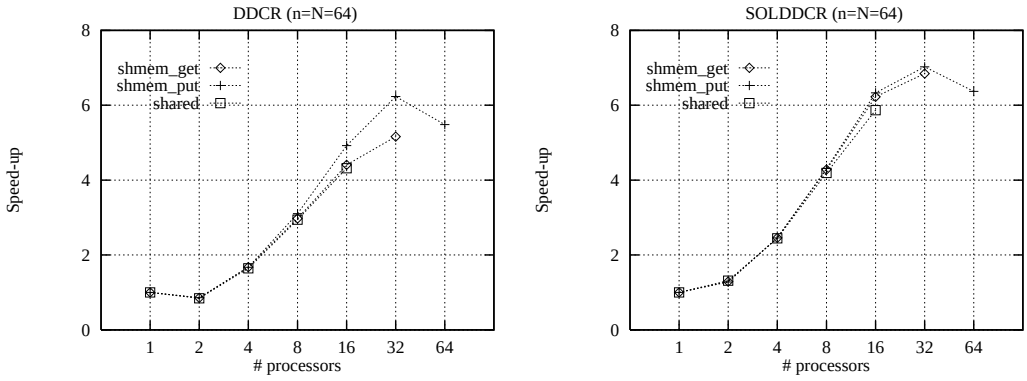

Figure 6: Speed-up for DDCR and SOLDDCR on the Cray T3D. Results of `PRIVATE` data implementation (`SHMEM_GET` and `SHMEM_PUT`) and `SHARED` data implementation.

be preferred, partly because the latency of the `SHMEM_PUT` is less than of `SHMEM_GET`, partly because data communication and floating point operations are performed concurrently. This can be realized by sending data at the moment it has been calculated rather than to wait until it is needed by other processors. To give an example, we have rewritten the first step of the cyclic reduction process as shown in Table 3. The third and fourth rule have been changed (cf. Table 7) and matrix e_0 is sent to its left neighbor, before matrix f is calculated. Consequently, data communication and floating point operations can be performed in parallel.

Figure 7 gives the speed-up of the preprocessing routine DDCR and the solution process routine SOLDDCR on the Cray T3D of the `SHMEM_PUT` implementation. Note, that for $N = p = 64$, pure cyclic reduction is applied and not the combination with domain decomposition, because each domain consists exactly of one block row. Note, that for the same block size, i.e., 48, the highest speed-up is achieved for the largest problem size. Analogously, for a fixed N_p, the number of diagonal blocks per processor (see Figure 8), the highest performance (> 1.2 Gflop/s) is achieved for the largest block size. For large problems, this is a very promising result.

7. Numerical Results of Algorithm 2

We have done experiments with Algorithm 2 on a Cray T3E in Delft, and used up to 64 processors. These experiments give us insight in the speed-up to be expected if we would use a machine with

p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7
$d = LU$		$d = LU$ $e = d^{-1}e$		$d = LU$ $e = d^{-1}e$		$d = LU$ $e = d^{-1}e$	
Send e to the left ($e(p_i) = e^R(p_{i-1})$).							
$f = d^{-1}f$		$f = d^{-1}f$		$f = d^{-1}f$		$f = d^{-1}f$	
Send f to the right ($f(p_i) = f^L(p_{i+1})$). BARRIER							
	$d = d - f.e^R$	$x = -e^R.e$	$d = d - f.e^R$	$x = -e^R.e$	$d = d - f.e^R$	$x = -e^R.e$	
Send x to the right ($x(p_i) = x^L(p_{i+1})$).							
	$d = d - e.f^L$	$y = -f^L.f$	$d = d - e.f^L$	$y = -f^L.f$	$d = d - e.f^L$	$y = -f^L.f$	$d = d - e.f^L$
Send y to the left ($y(p_i) = y^R(p_{i-1})$).							
$e_0 := e^R$	$e_0 := e^R$ $f_0 := f^L$	$e_0 := e^R$ $f_0 := f^L$	$e_0 := e^R$ $f_0 := f^L$	$e_0 := e^R$ $f_0 := f^L$	$e_0 := e^R$ $f_0 := f^L$	$e_0 := e^R$ $f_0 := f^L$	$f_0 := f^L$
BARRIER							
	$f := y^R$		$e := x^L$ $f := y^R$		$e := x^L$ $f := y^R$		$e := x^L$
End of the first step of the cyclic reduction process.							

Table 7: The improved version of the first cyclic reduction step for p=N=8. On the j-th processor, d, e and f correspond to d_j, e_j and f_j in (3.7), respectively. e_0 and f_0 are extra fill-in blocks which correspond to $\tilde{e}_{2j-1}$ and $\tilde{f}_{2j+1}$ in (3.8) and (3.9). x and y are auxiliary arrays used for data transfer. The superscripts ^{L} and ^{R} denote whether the matrix is received from a left or right processor, respectively.

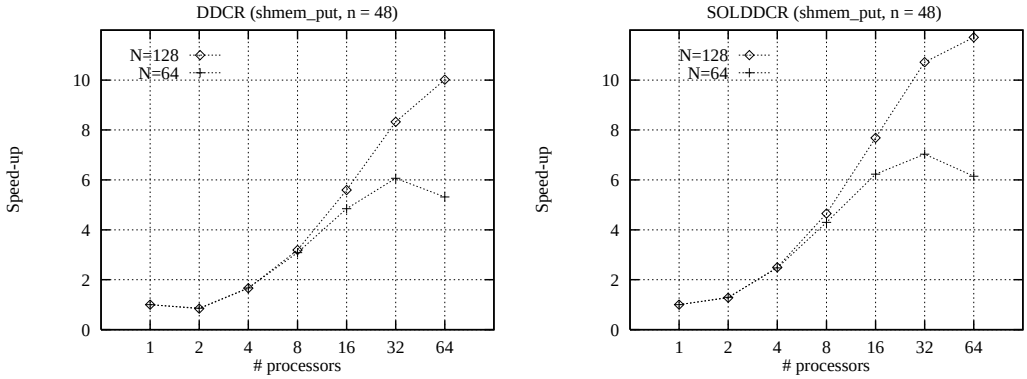

Figure 7: Speed-up for the SHMEM_PUT implementations of DDCR and SOLDDCR on the Cray T3D.

more processors.

Suppose that the maximum allowed value of the Krylov subspace m is much smaller than the size of the matrices A and B. If $A - \sigma B$ can be overwritten by its LU decomposition, the amount of memory required for the current sequential implementation is approximately $24\text{nnz}(A) + 16Nn(4 + 3(n + m))$ bytes, in which $\text{nnz}(A)$ is the number of nonzero entries in the matrix A. If we use the parallel DDCR preconditioner, the additional amount of memory is approximately $32n^2(N+3p)+0.2\text{nnz}(A)$ bytes. If $A - \sigma B$ cannot be overwritten by its LU decomposition, an extra $48Nn^2$ bytes of memory is required. Each processor of the machine in Delft contains 128 Mbytes of memory. Hence for the larger test cases, we have to use several processors. In order to study the obtained speed-ups, in Section 7.1 results for some smaller test cases are shown. In Section 7.2 we compare the performance obtained for a larger test case ($N = 320$ and $n = 128$) on a single (vector) processor of the Cray C90 with that on several processors of the Cray T3E.

In all experiments, at step 4 of Algorithm 2 we take it_{GMRES} equal to zero which implies that $\tilde{z} = r$. With this special choice, the JD algorithm is closely related to the Arnoldi method. From

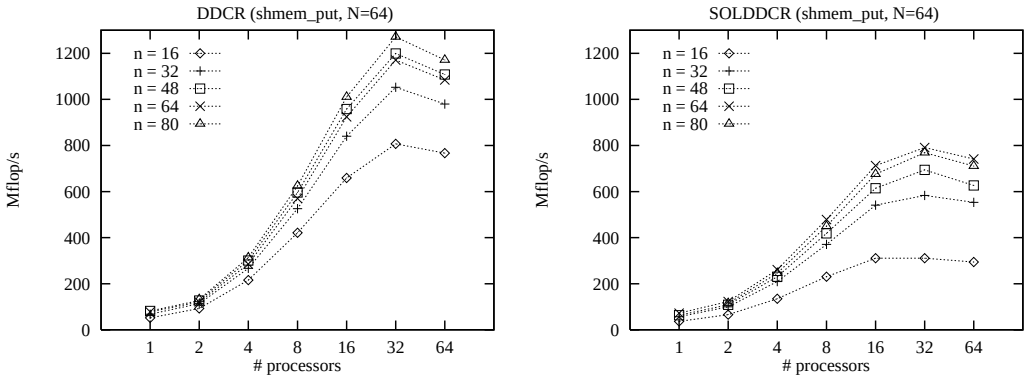

Figure 8: Mflop/s achieved on a Cray T3D for DDCR and SOLDDCR.

numerical experiments this appears to be the best choice for minimizing the total wall clock time. If we use some steps of GMRES, we can reduce the total number of steps of the JD method, but for our examples, this does not compensate for the extra work required for GMRES. The other parameters in Algorithm 2 were chosen as follows: $iter = 60$; $N_{ev} = 20$; $tol_{JD} = 10^{-6}$; $k_{min} = 10$; $m = 30$. Unless mentioned otherwise, no steps of iterative refinement (as explained in Section 4) are used.

7.1 Speed-ups on the Cray T3E

Table 8 shows the results for a small test problem generated by CASTOR, in which $n = 64$ and $N = 40$. After 60 iteration steps, we obtain 15 eigenpairs that satisfy the acceptance criterion. The second column shows the wall clock time measured in seconds required for the construction of the complete LU decomposition. The numbers in parentheses show estimates for the Mflop-rates. These numbers have been calculated by using the estimate (3.12) for the number of multiplications. [3] For small p we obtain a reasonable fraction of the theoretical peak performance, which is due to the level-3 `BLAS` routines that perform most of the work in the preprocessing phase. When the number of processors increases from one to two, the wall clock time increases. This is due to the computation of the extra fill-in blocks in L and U which are generated when the reordering technique for parallel processing is performed. Increasing the number of processors again with a factor two, approximately halves the wall clock time. Even for this small problem, the wall clock time for constructing L and U can be reduced by a factor of four by parallel processing. The third column shows the wall clock time required for the JD iteration without the time required for preprocessing. Since part of the algorithm is not performed in parallel (solution of the projected eigenvalue problem), again we cannot expect linear speed-up. However, we see a reduction of a factor of five in wall clock time. The fourth column shows the total time spent in performing the triangular solves. Again the numbers in parentheses show estimates for the Mflop-rates. These numbers have been calculated by using the estimate (3.13) for the number of multiplications. These Mflop-rates are significantly lower than the Mflop-rates obtained for the construction of L and U. The level-2 `BLAS` routines used for the triangular solves are significantly slower than the level-3 `BLAS` routines used for the construction of L and U, because the ratio of computations and memory-to-processor data transfer is much more favourable for level-3 `BLAS` than for level-2 `BLAS`.

We also performed similar calculations for a larger test problem in which $n = 64$ and $N = 160$. This problem is too large to run on one processor. The results are shown in Table 9. Again the estimated Mflop-rate is shown in parentheses. For small p the performance per processor is significantly smaller

[3] Suppose that one floating point operation (flop) denotes either a multiplication, a division, a subtraction, or an addition of two real variables. Since we use complex arithmetic, and since both in the preprocessing and in the solution process almost every multiplication can be combined with a subtraction or an addition, the number of flops is approximately equal to the number of multiplications multiplied by 8.

p	Preprocessing	Time JD without time for preprocessing	Total time triangular solves
1	0.87 (224)	6.57	2.30 (90)
2	1.10 (330)	4.04	1.75 (155)
4	0.57 (783)	2.30	0.93 (331)
8	0.32 (1541)	1.52	0.56 (578)
10	0.29 (1700)	1.49	0.55 (582)
20	0.22 (2350)	1.29	0.50 (672)

Table 8: Wall clock times in seconds on the Cray T3E for $N = 40$ and $n = 64$. The number of nonzero entries in the CASTOR matrices A and B is 196300 and 89196, respectively.

p	Preprocessing	Time JD without time for preprocessing	Total time triangular solves
2	5.72 (254)	26.86	11.68 (108)
4	2.88 (620)	12.24	5.91 (240)
8	1.49 (1316)	6.05	3.11 (481)
10	1.25 (1597)	5.34	2.66 (567)
16	0.82 (2488)	3.52	1.79 (859)
20	0.72 (2852)	3.11	1.57 (963)
32	0.51 (4055)	2.44	1.20 (1289)

Table 9: Wall clock times in seconds on the Cray T3E for $N = 160$ and $n = 64$. The number of nonzero entries in the CASTOR matrices A and B is 798632 and 363228, respectively.

than that obtained for the test problem mentioned above. However, this improves if p increases, because then the same amount of data is distributed over more processors. This is probably a cache effect.

7.1.1 Analysis of the speed-ups In the following, we will see how well the results of Tables 8 and 9 agree with what we may expect for such relatively small test problems. The computations in the JD algorithm can be divided into three parts: the solution of the (small) projected eigenvalue problems, the triangular solves with L and U, and a part consisting of matrix-vector multiplications with B, inner products and vector updates. Suppose that the CPU times for execution on one processor for these parts are denoted by t_{seq}, t_{LU}, and t_{linpar}, respectively. If we increase the order of A and B, t_{seq} hardly changes, since we keep the size of the projected systems fixed. However, both t_{LU} and t_{linpar} increase with the order of the eigenvalue problem. Hence for large eigenproblems, t_{seq} is very small compared with t_{LU} and t_{linpar}. If we assume that the time for communication can be neglected and the inner products scale linearly, the expected speed-up S_{JD} when performing the JD algorithm on p processors instead of on one processor, is

$$S_{JD} = \frac{t_{seq} + t_{LU} + t_{linpar}}{t_{seq} + \frac{t_{LU}}{S_{LU}} + \frac{t_{linpar}}{p}}, \tag{7.1}$$

in which S_{LU} is the expected speed-up for the triangular solves. In the following, we will show that an approximation of S_{LU} is given by

$$S_{LU} = \frac{3pN}{(5 - \frac{2}{p})(N + 1 - p) + 5p(^2\log p - 1)}. \tag{7.2}$$

On just one processor, the standard LU approach is applied and in that case the number of multiplications in the triangular solves is approximately $3Nn^2$. Almost every multiplication can be combined with an addition, hence the wall clock time required on one processor is approximately $3Nn^2t_{mul}$, in which t_{mul} is the time required for performing a complex multiplication combined with an addition. On p processors, the parallel DDCR preconditioner is used. The first part of the triangular solves that corresponds with the elimination of y_1 from (3.4) scales linearly. The number of multiplications in this part is approximately $3(N-p+1)n^2\omega$, in which ω is the number of multiplications in the DDCR approach divided by the number of multiplications in the standard LU approach. From (3.13) it follows that $\omega \approx (5-\frac{2}{p})/3$. Hence the wall clock time required for this first part based on domain decomposition is approximately $(3\omega(N-p+1)n^2t_{mul})/p$. The second part of the triangular solves is the solution of (3.6) with cyclic reduction. The wall clock time required for this part is approximately $5n^2t_{mul}$ multiplied by the number of steps in cyclic reduction (approximately $^2\log p - 1$). Equation (7.2) is obtained by dividing the approximate wall clock time on one processor by the sum of the wall clock times required for the first and second part of the triangular solves on p processors.

Figure 9 shows both the predicted speed-up by (7.1) for $N = 40$ and the measured speed-up obtained from Table 8. From the results it appears that the predicted speed-up corresponds quite well with the measured speed-up, although it is a little too high for $p = 20$. This is caused by the fact that the simple prediction model assumes that the inner products scale linearly, which is not quite true.

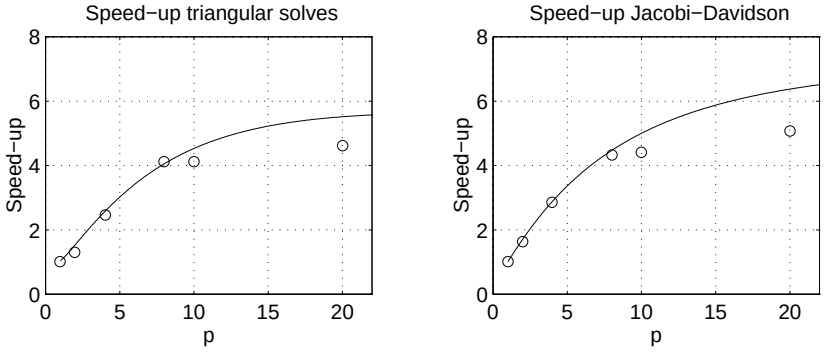

Figure 9: The predicted (straight line) and measured speed-up ('$\circ$') of the JD algorithm for $N = 40$ and $n = 64$ on a Cray T3E. Measured speed-up is shown for p=1, 2, 4, 8, 10, and 20.

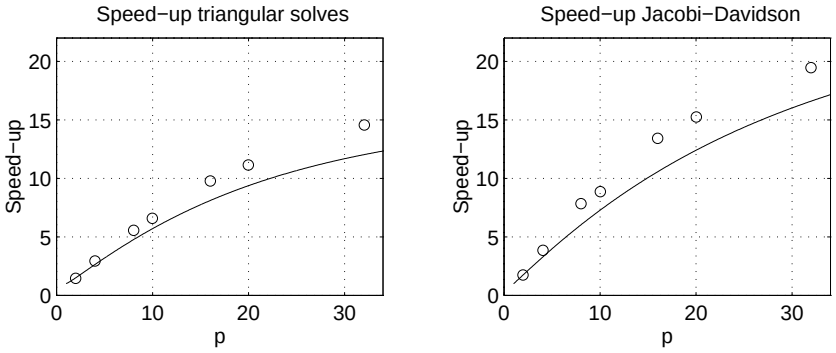

Figure 10: The predicted (straight line) and measured speed-up ('$\circ$') of the JD algorithm for $N = 160$ and $n = 64$ on a Cray T3E. Measured speed-up is shown for p=2, 4, 8, 10, 16, 20, and 32.

Figure 10 again shows both the predicted and measured speed-up for the test case in which $n = 64$ and $N = 160$. It appears that the measured speed-up is higher than what the simple model predicts. As mentioned before, this is caused by the fact that when the amount of data per processor decreases, the performance per processor can increase (probably due to cache effects). Our simple prediction model does not account for this effect.

7.2 One processor of the C90 compared with several processors of the T3E

In this section, we do experiments with a larger test case in which $N = 320$ and $n = 128$. First, we show the results obtained on one processor of the C90. Figure 11 shows the convergence behaviour of

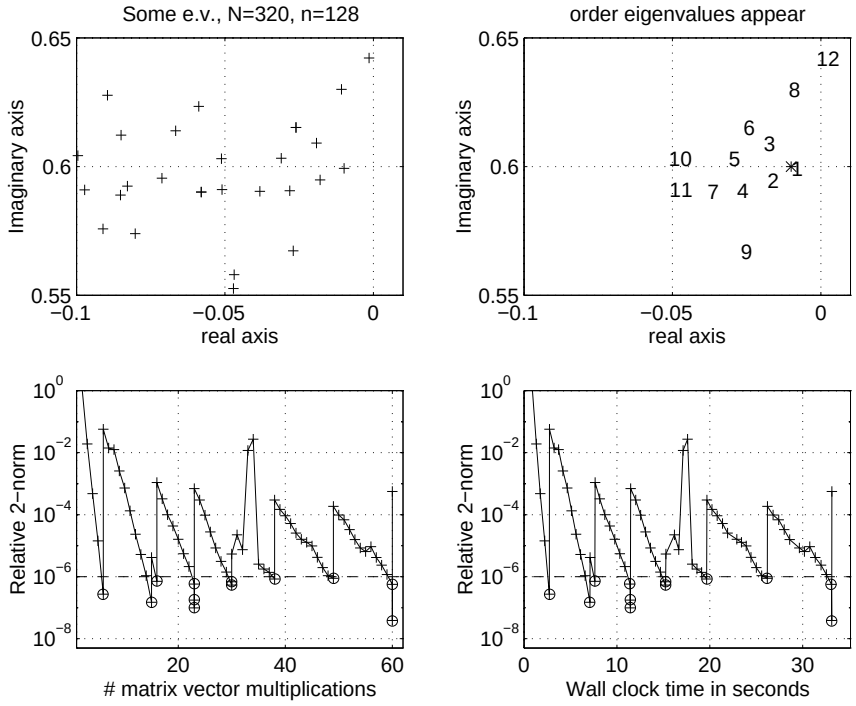

Figure 11: Numerical results of Algorithm 2 applied to (4.1) obtained on one processor of the C90. $N = 320$, $n = 128$.

Algorithm 2 applied to (4.1). Because we use only one processor, the sequential standard LU approach is used instead of the DDCR method. The matrix B is stored in Jagged Diagonal Storage format, which exploits the sparsity pattern and gives vectorizable code in the matrix-vector multiplication. After 60 iteration steps, we find 12 eigenvalues. In the upper-left corner of Figure 11, some of the eigenvalues are shown, and the picture in the upper-right corner shows the sequence in which they appear when the target σ is chosen as is indicated by a '$*$' in the figure. The eigenvalues appear in the order that corresponds with the lowest distance to the target. The lower-left corner shows the relative 2-norm of the residual, viz. $\|Qx - \theta x\|_2/|\theta|$, against the number of matrix-vector multiplications with the matrix $Q = (LU)^{-1}B$. Each '$\circ$' indicates a converged eigenvalue. It can happen that at a certain step of the JD method, more than one new eigenvalue is found that satisfies the acceptance criterion. In this example, σ is very close to an eigenvalue. The algorithm convergences very quickly to this eigenvalue. From the results it appears that the first 8 eigenpairs are found quickly, but for the last 4 eigenpairs more iteration steps are required. Hence for this example, it may be a good idea to adapt the target after 30 JD steps have been performed. The picture in the lower-right corner shows the same relative 2-norm against the wall clock time required on one processor of the C90. This time does

not include the 17.5 seconds required for the construction of L and U.

The wall clock times for preprocessing, for the 60 iteration steps of Algorithm 2, and for the triangular solves are shown in Table 10. Again the numbers in parentheses show the estimated Mflop-rates. The first line shows the results obtained without iterative refinement as explained in Section 4, the second line shows the results obtained with one step of iterative refinement. This does not change the computed eigenvalues significantly: the relative difference is smaller than 10^{-6}. We notice that

iterative refinement?	Preprocessing	Time JD without time for preprocessing	Total time triangular solves
No	17.5 (716)	30.3	11.5 (656)
Yes	17.5 (716)	50.0	23.0 (656)

Table 10: Wall clock times in seconds on one processor of the Cray C90. $N = 320$, $n = 128$. The number of nonzero entries in A and B is 6491568 and 2886552, respectively.

the obtained Mflop-rates are not far from the theoretical peak performance of 960 Mflop/sec. Our implementation of Algorithm 2 vectorizes very well thanks to the use of `BLAS` routines whenever possible. Moreover, the use of complex arithmetic gives a better ratio of floating point operations and data transfer than real arithmetic. In order to reduce the effect of memory bank conflicts during the construction of L and U, it is very important that the leading dimension of the arrays that are passed as arguments in the `BLAS` routines do not contain high powers of two. Therefore, the leading dimension of such arrays has been increased by one, resulting in a reduction of the time required for the construction of L and U from 54 seconds to 17.5 seconds.

On the Cray T3E in Delft Algorithm 2 is applied in combination with the parallel DDCR preconditioner. We used the same test problem and the same parameters for Algorithm 2 as described above. Due to the limited amount of memory per processor at least 10 PEs are required to solve this problem. Of course, we find the same eigenvalues as on one processor of the C90. The difference with the eigenvalues found on the C90 is approximately 10^{-6}, which is in agreement with the acceptance criterion. The convergence behaviour depends slightly on the number of processors. This is illustrated by Figure 12, which shows the relative 2-norm of the residual against the wall clock time measured on 10, 20, 32, and 64 processors.

The wall clock times for the preprocessing, the 60 iteration steps of Algorithm 2, and for the triangular solves are shown in Table 11. Again the numbers in parentheses show estimates for the Mflop-rates. From the results it appears that for this test case the code parallelizes well. The wall

p	Preprocessing	Time JD without time for preprocessing	Total time triangular solves
10	20.61 (1546)	32.62	19.2 (628)
16	14.15 (2308)	19.92	12.5 (978)
20	11.17 (2949)	16.95	10.8 (1144)
32	7.85 (4249)	10.9	7.4 (1682)
64	4.68 (7180)	7.1	5.1 (2454)

Table 11: Wall clock times in seconds on the Cray T3E. $N = 320$, $n = 128$.

clock times on 10 processors of the T3E are comparable with the wall clock time on one processor of the C90.

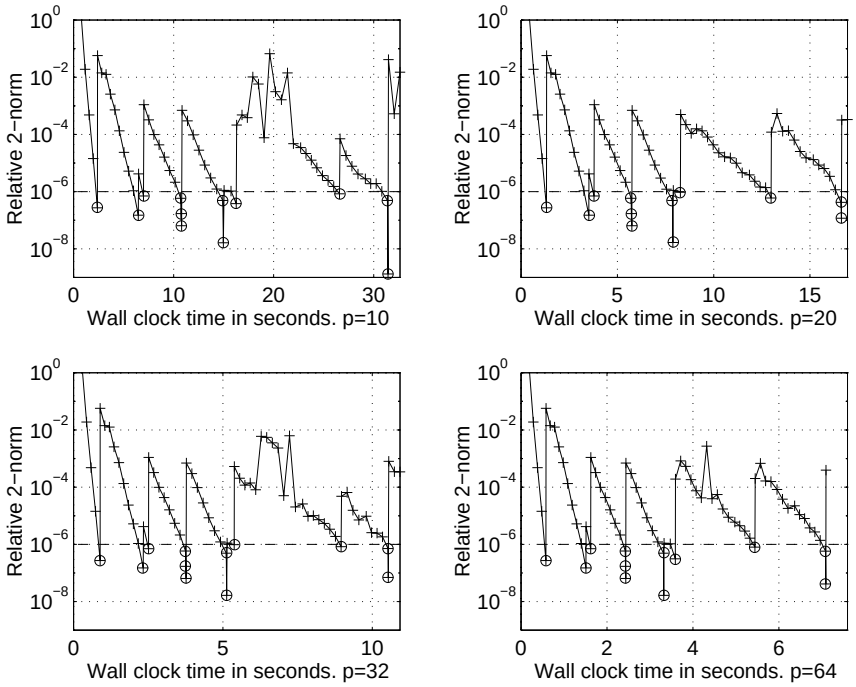

Figure 12: Numerical results of Algorithm 2 applied to (4.1) obtained on p processors of the Cray T3E. The pictures show the relative 2-norm of the residual against the wall clock time measured in seconds. $N = 320$, $n = 128$.

8. Conclusions

We have studied the Jacobi-Davidson method for the parallel computation of a few selected eigenvalues of large generalized eigenvalue problems arising in the stability investigation of tokamak plasmas. The JD method has been combined with several preconditioning techniques. For our applications, the SPAI preconditioner of Grote and Huckle was not very successful because we did not succeed in approximating the inverse of $A - \sigma B$ by a sparse matrix. The ILUT preconditioner of Saad is not suited for our applications either, since the subblocks in the factors L and U are not sparse enough to take profit from it. A complete block LU decomposition like the standard LU approach appears to be a proper preconditioner. This approach is much more robust than a point-wise ILU-type preconditioner because pivoting is used. However, in order not to disturb the block structure of the matrix, the search for pivot elements is restricted to the blocks on the main diagonal. The numerical experiments performed on one processor of the C90 demonstrate that this method is well-vectorizable.

In order to improve possibilities for parallelization, the block rows and block columns of $A - \sigma B$ are reordered simultaneously before the LU decomposition of $A - \sigma B$ is performed. The reordering is based on a combination of domain decomposition and cyclic reduction. Numerical results performed on a Cray T3D and a Cray T3E demonstrate that the resulting DDCR preconditioner parallelizes well: both the time for the construction of L and U and for performing the triangular solves hardly increase when the number of processors increases with the same factor as the number of diagonal blocks.

Most other ingredients in the Jacobi-Davidson method like the matrix-vector multiplications, vector updates and inner products parallelize very well. Only the construction and solution of the small projected eigenvalue problems in the Jacobi-Davidson method does not parallelize. However, the size of these systems is kept small and is independent of the problem size. Hence for large applications, the total time spent in solving the projected systems is negligible (less than 0.5 seconds for the numerical

experiments described in Section 7), so that our implementation of Algorithm 2 parallelizes well, which is demonstrated by the results shown in Section 7.

If we consider only the wall clock times required for computing several eigenvalues, we conclude that 10 processors of the Cray T3E give approximately the same computational power as one processor of the C90. As a consequence, the Cray T3E can solve the eigenvalue problems faster than the Cray C90, if a sufficient number of processors is used.

The main reason to study implementations on distributed memory machines like the T3E is the memory bound of shared memory machines. The amount of memory required for the current parallel implementation of Algorithm 2 for large n and N ($n \gg m$ and $N \gg p$) is approximately $180Nn^2$ bytes. The current sequential implementation requires approximately $130Nn^2$ bytes. The amount of memory per processor on the Cray T3E at HPαC, Delft is 128 Mbytes and the total amount of memory of the 80-processors machine is approximately 10 Gbytes. This is only slightly larger than the 8 Gbytes of main memory of the C90 at SARA, Amsterdam. Hence with the current Delft's configuration, we cannot solve larger problems than on the C90. However, processors of the T3E can have up to 2 Gbytes of memory. Moreover, the number of processors can be extended to 2048, so the total amount of memory can be as large as 4 Tbytes. This makes the machine perfectly suitable for solving large eigenvalue problems like those coming from CASTOR. On such machines with 4 Tbytes of memory, we could calculate the Alfvén spectrum for eigenvalue problems with $N = 8000$ diagonal blocks of size $n = 1600$. For such a large application, a rough estimate of the total wall clock time required for performing the preprocessing and 60 iteration steps of Algorithm 2, like we did in our experiments, is five hours. About 90% of this time would be required for the construction of the factors L and U. This estimate is based on the numerical results and performance model described in Section 7.

Acknowledgements

The authors wish to thank Herman te Riele for many stimulating discussions and numerous suggestions for improving the presentation of the paper. They also thank Gerard Sleijpen and Henk van der Vorst for carefully reading the paper and suggesting several improvements. Further they like to thank Ronald van Pelt of Cray Research, The Netherlands, for his programming advices for both the Cray T3D and Cray T3E, and HPαC (Delft) and SARA (Amsterdam) for their technical support. They gratefully acknowledge Cray Research for a sponsored account on the Cray T3D, and the Dutch National Computing Facilities Foundation NCF for the provision of computer time on the Cray C90 and the Cray T3E.

References

1. Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the solution of linear systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, 1994. http://www.netlib.org/templates/.
2. Albert Booten and Henk van der Vorst. Cracking Large-Scale Eigenvalue Computations, Part I: Algorithms. *Computers in Physics*, 10(3):239–242, 1996.
3. Albert Booten and Henk van der Vorst. Cracking Large-Scale Eigenvalue Computations, Part II: Implementations. *Computers in Physics*, 10(4):331–334, 1996.
4. J.G.C. Booten, D. Fokkema, G.L.G. Sleijpen, and H.A. van der Vorst. Jacobi-Davidson methods for generalized MHD-eigenvalue problems. *Zeitschrift für angewandte Mathemathik und Mechanik 76, Supplement 1*, pages 131–134, 1996.
5. Diederik R. Fokkema, Gerard L.G. Sleijpen, and H.A. van der Vorst. Jacobi-Davidson style QR and QZ algorithms for the partial reduction of matrix pencils. Technical Report nr. 941, Department of Mathematics, University Utrecht, January 1996. revised: 1997 (to appear in SISC).

6. Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore and London, third edition, 1996.

7. M.J. Grote and T. Huckle. Parallel Preconditioning with Sparse Approximate Inverses. *SIAM J. Sci. Comput.*, 18(3):838–853, 1997.

8. D. Heller. Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems. *SIAM J. Numer. Anal.*, 13:484–496, 1978.

9. W. Kerner, S. Poedts, J.P. Goedbloed, G.T.A. Huysmans, B. Keegan, and E. Schwartz. In P. Bachman and D.C. Robinson, editors, *Proceedings of 18th Conference on Controlled Fusion and Plasma Physics*. EPS: Berlin, 1991. IV.89-IV.92.

10. Y. Saad. ILUT: a dual threshold incomplete LU factorization. *Num. Lin. Alg. Appl.*, 1:387–402, 1994.

11. G.L.G. Sleijpen, J.G.L. Booten, D.R. Fokkema, and H.A. van der Vorst. Jacobi-Davidson Type Methods for Generalized Eigenproblems and Polynomial Eigenproblems. *BIT*, 36:595–633, 1996.

12. G.L.G Sleijpen and H.A. van der Vorst. A Jacobi-Davidson iteration method for linear eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 17(2):401–425, april 1996.

13. A. van der Ploeg. Reordering Strategies and LU-decomposition of Block Tridiagonal Matrices for Parallel Processing. Technical Report NM-R9618, CWI, Amsterdam, October 1996.

14. Aad J. van der Steen. *Benchmarking of High Performance Computers for scientific and technical computation*. PhD thesis, Utrecht University, 1997.

A. Appendix

In Table 12 the number of matrix-matrix multiplications (MM), complete LU decompositions (LU) on n-by-n-blocks, systems solutions after LU factorizations (TR), and n-by-n-block updates (BU) are listed. The counts MM and TR are split into a domain decomposition part (DD) and a cyclic reduction part (CR). In this table p denotes the number of processing elements and N_p is the number of main diagonal blocks per processor. For simplicity we have taken $N = N_p \times p$. Analogous functions for the solution process which consists of matrix-vector multiplications (MV), n-vector updates (VU) and the solution of two triangular equations (SOLVE) are listed in Table 13.

$$
\begin{array}{ll}
\text{MM(DD)} & = \begin{cases} N_p - 1 = N - 1 & \text{if } p = 1 \\ (N_p - 1)(4p - 3) - 2 & \text{otherwise} \end{cases} \\[2ex]
\text{MM(CR)} & = \begin{cases} 0 & \text{if } p = 1, 2 \\ 2\sum_{i=1}^{^2\log p}(\lceil \frac{p}{2^i} \rceil - 1) + 4\sum_{i=3}^{^2\log p}(\lceil \frac{p}{2^i} \rceil - 1) & p \geq 4 \\ \sum_{i=1}^{^2\log p} \lceil \frac{p}{2^i} \rceil + 3\sum_{i=1}^{^2\log p - 1}(\lceil \frac{p}{2^i} \rceil - 1) & N = p \end{cases} \\[2ex]
\text{LU} & = p \times N_p = N \\[2ex]
\text{TR(DD)} & = (N_p - 1)(2p - 1) + 1 \\[2ex]
\text{TR(CR)} & = \begin{cases} 2\sum_{i=1}^{^2\log p}(\lceil \frac{p}{2^{i-1}} \rceil - 1) & \text{if } N > p \\ 2\sum_{i=1}^{^2\log p} \lceil \frac{p}{2^i} \rceil + \sum_{i=1}^{^2\log p - 1}(\lceil \frac{p}{2^i} \rceil - 1) & \text{otherwise} \end{cases} \\[2ex]
\text{BU} & = \begin{cases} 0 & \text{if } N_p = 1 \\ p - 1 & \text{otherwise} \end{cases}
\end{array}
$$

Table 12: Number of block operations performed by `BLAS` and `LAPACK` routines in DDCR.

$$
\begin{array}{ll}
\text{MV(DD)} & = (N_p - 1)(4p - 2) \\[2ex]
\text{MV(CR)} & = \begin{cases} 4\sum_{i=2}^{^2\log p}(\lceil \frac{p}{2^{i-1}} \rceil - 1) & N > p \\ 2\sum_{i=1}^{^2\log p} \lceil \frac{p}{2^i} \rceil + 2\sum_{i=1}^{^2\log p - 1}(\lceil \frac{p}{2^i} \rceil - 1) & N = p \end{cases} \\[2ex]
\text{SOLVE} & = p \times N_p = N \\[2ex]
\text{VU} & = \begin{cases} 0 & \text{if } N_p = 1 \\ p - 1 & \text{otherwise} \end{cases}
\end{array}
$$

Table 13: Number of block operations performed by `BLAS` and `LAPACK` routines. N is the number of main diagonal blocks; N_p the number of main diagonal blocks per processor and p the number of processors.