FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Data Distribution and Communication Schemes for Solving Sparse Systems of Linear Equations from FE Applications by Parallel CG Methods

Achim Basermann

KFA-ZAM-IB-9323

September 1993
(Stand 01.10.93)

Tagungsband des Workshops über Parallelverarbeitung, 20.-24. September, Lessach, Österreich

Data Distribution and Communication Schemes for Solving Sparse Systems of Linear Equations from FE Applications by Parallel CG Methods

A. Basermann [a]
[a] Central Institute for Applied Mathematics
Research Centre Jülich GmbH, 52425 Jülich, Germany
email: A.Basermann@kfa-juelich.de

Abstract

For the solution of discretized ordinary or partial differential equations it is necessary to solve systems of equations with coefficient matrices of different sparsity pattern, depending on the discretization method; using the finite element (FE) method results in largely unstructured systems of equations. Iterative solvers for equation systems mainly consist of matrix-vector products and vector-vector operations. A frequently used iterative solver is the method of conjugate gradients (CG) with different preconditioners. For parallelizing this method on a multiprocessor system with distributed memory, in particular the data distribution and the communication scheme depending on the used data structure for sparse matrices are of greatest importance for the efficient execution. These schemes can be determined before the execution of the solver by preprocessing the symbolic structure of the sparse matrix and can be exploited in each iteration. In this report, data distribution and communication schemes are presented which are based on the analysis of the column indices of the non-zero matrix elements. Performance tests of the developed parallel CG algorithms have been carried out on the distributed memory system INTEL iPSC/860 of the Research Centre Jülich with sparse matrices from FE models. These methods have performed well for matrices of very different sparsity pattern.

Keywords: Sparse matrices; Finite element method; Conjugate gradients method; Parallelization; Distributed memory computer; Data distribution; Communication scheme.

1 Introduction

For the solution of discretized ordinary or partial differential equations it is necessary to solve systems of equations with coefficient matrices of different spar-

sity patterns, depending on the discretization method; using the finite element method (FE) results in largely unstructured systems of equations.

Iterative methods for solving linear systems mainly consist of matrix-vector products and vector-vector operations; the main work in each iteration is usually the computation of matrix-vector products. Therein, accessing the vector is determined by the sparsity pattern and the storage scheme of the matrix.

A frequently used iterative solver is the method of conjugate gradients (CG) with different preconditioners [10] [13]. In 1990, Aykanat e.a. presented a modified CG algorithm [4] with better parallelization properties than the original method developed by Hestenes and Stiefel.

For parallelizing iterative solvers on a multiprocessor system with distributed memory, in particular the data distribution and the communication scheme depending on the data structures used for sparse matrices are of greatest importance for the efficient execution. In this context, different reordering strategies of the sparse matrix have been investigated to reduce waiting times by performing communication and computation overlapped. Additionally, the reverse Cuthill-McKee scheme [16] is applied to diminish the bandwidth of the matrix. Depending on the sparsity pattern of the matrix, bandwidth reduction results in a considerable decrease of communication. The data distribution and the communication scheme are determined before the execution of the solver by preprocessing the symbolic structure of the sparse matrix and are exploited in each iteration. Moreover, the schemes are applicable as long as the sparsity pattern of the matrix which is determined by the discretization mesh does not change, i.e. they can be used in each time step of a time dependent problem or in each iterative step of a nonlinear problem which is solved by linearization. In this report, data distribution and communication schemes are presented which are based on the analysis of the column indices of the non-zero matrix elements.

Performance tests of the developed parallel CG algorithms with preconditioning have been carried out on the distributed memory system INTEL iPSC/860 of the Research Centre Jülich with sparse matrices from two FE models. The first FE model comes from environmental science; it simulates the behaviour of pollutants in geological systems [1] [17]. In the second FE model from structural mechanics, stresses in materials induced by thermal expansion are calculated by applying the FE program SMART [2].

2 The Method of Conjugate Gradients

The method of conjugate gradients [10] is an algorithm for solving systems of linear equations $Ax = b$, particularly for sparse coefficient matrices A. The method converges for matrices which are symmetric and positive definite.

Aykanat e.a. [4] suggested a modified CG algorithm (see algorithm 2.1) which has better parallelization properties than the original method.

Algorithm 2.1. The modified CG method

Choose an arbitrary $x_0 \in \mathbb{R}^n$;

$$\begin{aligned} g_0 &= Ax_0 - b \\ d_0 &= -g_0 \end{aligned}$$

$i = 0, 1, \ldots$

$$\begin{aligned} \gamma_i &= \frac{g_i^T g_i}{d_i^T A d_i} \\ \delta_i &= \frac{\gamma_i (Ad_i)^T A d_i}{d_i^T A d_i} - 1 \\ g_{i+1}^T g_{i+1} &= \delta_i g_i^T g_i \\ x_{i+1} &= x_i + \gamma_i d_i \\ g_{i+1} &= g_i + \gamma_i A d_i \\ d_{i+1} &= -g_{i+1} + \delta_i d_i \end{aligned}$$

until $\|g_{i+1}\|_2 \leq \epsilon_r$.

In each iteration, the vectors x_i, g_i, and d_i are computed. x_i approximates the solution vector, g_i is the residue; d_i determines the direction in which the next approximation of the solution vector is searched for. The main work in each iteration consists in the computation of the matrix-vector product Ad_i. Furthermore, two dot products and three vector additions have to be performed. Iteration is continued until the euclidean norm of the residue is less than or equal to ϵ_r. Another stopping criterion which uses the maximum scaled absolute difference of the components of the latest two approximations of the solution vector is determined as follows:

$$\max_{j=1,\ldots,n} 2\frac{|x_{i+1}^j - x_i^j|}{|x_{i+1}^j| + |x_i^j|} \leq \epsilon_s. \tag{1}$$

The main difference between the original and the modified CG algorithm is that in the modified one all dot products are computed directly one after another without any other operations between. If each iteration is performed in parallel on a distributed memory system the local values of the dot products can be included in one message for determining the global values.

In the investigations, algorithm 2.1 has been performed with and without diagonal scaling [13], a simple preconditioner, which hardly contributes to the total execution time but usually accelerates the convergence considerably.

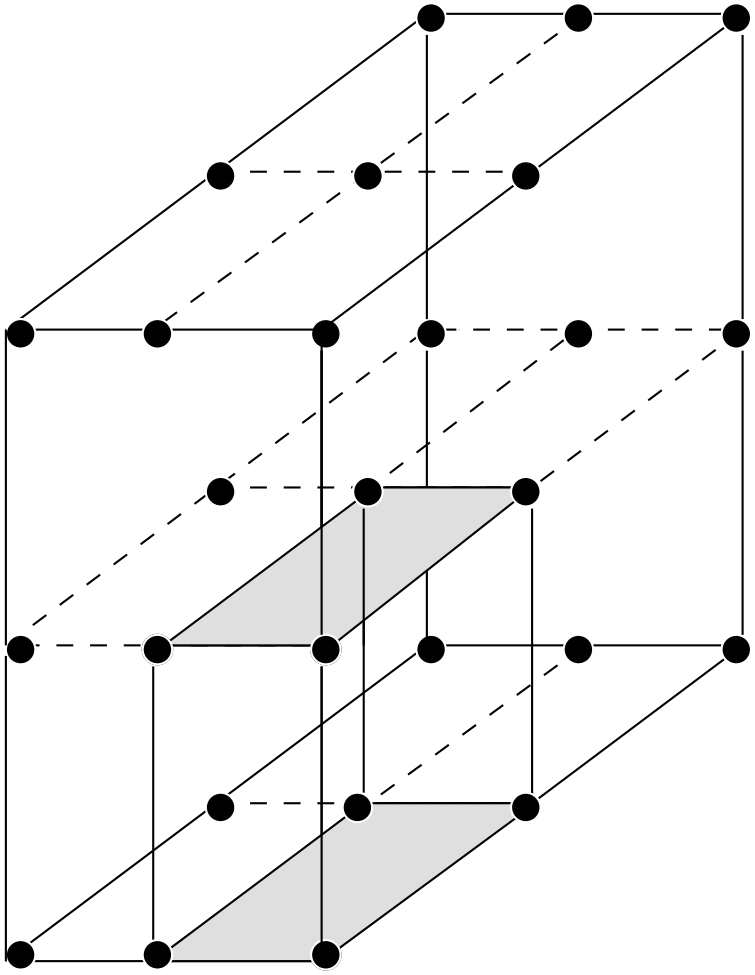

Figure 1: FE discretization mesh

3 Storage Schemes

Storage schemes for large sparse matrices depend on the sparsity pattern of the matrix, the considered algorithm, and the architecture of the computer system used. In the literature, many variants of storage schemes can be found [7] [8] [11] [12] [14] [15].

In FE models, the maximum number of non-zeros per row of the matrix is given by the geometry and the choice of the elements. The discretization mesh in figure 1 e.g. consists of hexahedron elements with nodal points in each corner. The number of rows of the coefficient matrix is given by the number of nodes, the number of non-zeros per row by the number of nearest neighbours of a node. The node in the middle of the mesh e.g. has eight neighbours in the middle plane, nine in the plane below, and nine in the plane above, totally 26 nearest neighbours. Therefore, the corresponding row of the matrix has 27 non-zeros. Boundary nodes

have less than 26 nearest neighbours. More nearest neighbours occur in meshes which consist of more complicated elements, e.g. octahedrons. Furthermore, different elements in one mesh, more nodes per element and a finer discretization in parts of a mesh are possible for the FE discretization. Additionally, the number of free degrees per node can increase if there are rotation axes e.g. besides the three spatial directions. Therefore, the number of non-zeros per row varies considerably for irregular dicretization meshes.

In the following, two storage schemes for sparse matrices are presented; these schemes are frequently used in FE programs.

In the first case, the matrix is stored row-wise in two-dimensional arrays; this storage scheme is applied in [1]. The scheme is elucidated for matrix (2) in (3).

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 3 & 10 & 0 & 0 & 0 & 0 \\ 0 & 0 & \underline{10} & 4 & 11 & 14 & 12 & \underline{18} \\ 0 & 0 & 0 & 11 & 5 & 0 & 17 & 0 \\ 0 & 0 & 0 & 14 & 0 & 6 & 15 & 0 \\ 0 & 0 & 0 & 12 & 17 & 15 & 7 & 0 \\ 0 & 0 & 0 & 18 & 0 & 0 & 0 & 8 \end{pmatrix} \tag{2}$$

$$A^w = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 9 & 2 & 0 & 0 & 0 & 0 \\ 10 & 9 & 3 & 0 & 0 & 0 \\ 10 & 4 & \underline{18} & 14 & 12 & 11 \\ 11 & 5 & 17 & 0 & 0 & 0 \\ 15 & 14 & 6 & 0 & 0 & 0 \\ 12 & 17 & 7 & 15 & 0 & 0 \\ 18 & 8 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad A^s = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & 0 & 0 & 0 & 0 \\ 4 & 2 & 3 & 0 & 0 & 0 \\ 3 & 4 & 8 & 6 & 7 & 5 \\ 4 & 5 & 7 & 0 & 0 & 0 \\ 7 & 4 & 6 & 0 & 0 & 0 \\ 4 & 5 & 7 & 6 & 0 & 0 \\ 4 & 8 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{3}$$

The matrix A is stored in the two-dimensional arrays A^w and A^s. In principle, the non-zeros of A are shifted to the left. A^w contains the values of the non-zeros, A^s the corresponding column indices. The value 18 e.g. is in A in row 4 and column 8. The order of the matrix elements per row in A^w and A^s is different from that in matrix A since this is usually the case in FE programs caused by the assembly of the coefficient matrix from the single elements. In (3), many zeros are stored in A^w and A^s because the number of non-zeros per row varies considerably. For irregular discretization meshes, the storage requirements of this scheme are much higher than necessary.

This disadvantage is avoided by storing merely the non-zeros row-wise in three one-dimensional arrays. The storage scheme considered here can be found in similar form in e.g. [11]. The principle of the scheme is shown in (4) for matrix (2).

$$
\begin{aligned}
a^w &= (1 \mid 9\ 2 \mid 10\ 9\ 3 \mid \underline{10}\ 4\ 18\ 14\ 12\ 11 \mid 11\ 5\ 17 \mid 15\ 14\ 6 \mid 12\ 17\ 7\ 15 \mid 18\ 8), \\
a^s &= (1 \mid 3\ 2 \mid 4\ 2\ 3 \mid 3\ 4\ 8\ 6\ 7\ 5 \mid 4\ 5\ 7 \mid 7\ 4\ 6 \mid 4\ 5\ 7\ 6 \mid 4\ 8), \qquad (4) \\
a^z &= (1\ 2\ 4\ 7\ 13\ 16\ 19\ 23\ 25).
\end{aligned}
$$

The non-zeros of matrix A are stored row-wise in three one-dimensional arrays. a^w contains the values of the non-zeros, a^s the corresponding column indices. In a^z, the position of the beginning of each row in a^w and a^s is stored. The subdivisions in a^w and a^s have been added to mark the beginning of a new row. The value 10 e.g. is in matrix A in column 3 and row 4. This scheme is suitable for regular as well as for irregular discretization meshes and has usually less storage requirements than the former. Therefore, merely this scheme is applied in the following considerations.

4 Parallelization

4.1 Data Distribution

For parallelizing algorithm 2.1 on a distributed memory system, the matrix and vector arrays must be suitably distributed to each processor. For the considered data distribution schemes, the arrays a^w and a^s are distributed row-wise; the rows of each processor succeed one another. The distribution of the vector arrays corresponds component-wise to the row distribution of the matrix arrays.

Criteria for the data distribution can be: each processor gets the same number of rows or so many rows that each processor has nearly the same number of non-zeros. The number of operations for the computation of the matrix-vector product is proportional to the number of non-zeros; the remaining vector operations of one iteration are proportional to the number of rows. Another criterion is that each processor has to compute nearly the same number of operations. If the discretization mesh is regular, i.e. the sparsity pattern of the coefficient matrix is regular, all three criteria result in nearly the same data distribution. If the mesh is very irregular, the three distributions differ considerably.

The first case, i.e. each processor gets nearly the same number of rows, will be explained in (5) by distributing the array a^w from (4) to four processors. The distribution of the remaining arrays ensues analogously.

$$
\begin{aligned}
\text{Processor 0: } a_0^w &= (1 \mid 9\ 2) \\
\text{Processor 1: } a_1^w &= (10\ 9\ 3 \mid 10\ 4\ 18\ 14\ 12\ 11) \qquad (5) \\
\text{Processor 2: } a_2^w &= (11\ 5\ 17 \mid 15\ 14\ 6) \\
\text{Processor 3: } a_3^w &= (12\ 17\ 7\ 15 \mid 18\ 8)
\end{aligned}
$$

In the second case, a^w is distributed according to the criterion "each processor gets the same number of non-zeros", see (6).

$$\begin{array}{lrcl} \text{Processor 0:} & a_0^w & = & (1 \mid 9\ 2 \mid 10\ 9\ 3) \\ \text{Processor 1:} & a_1^w & = & (10\ 4\ 18\ 14\ 12\ 11) \\ \text{Processor 2:} & a_2^w & = & (11\ 5\ 17 \mid 15\ 14\ 6) \\ \text{Processor 3:} & a_3^w & = & (12\ 17\ 7\ 15 \mid 18\ 8) \end{array} \tag{6}$$

In the third case, i.e. each processor has to compute nearly the same number of operations, processor k, $k = 0, \ldots, p-1$, gets so many rows until

$$\frac{e_k + \xi n_k}{e + \xi n} \geq \frac{1}{p}, \quad \text{for } e_k, n_k \gg 10 \tag{7}$$

is satisfied for the first time, i.e. for the least number of rows possible. The row distribution is determined by analyzing the array a^z. p is the number of the processors used, e_k the number of non-zeros, and n_k the number of rows of processor k. e is the total number of non-zeros and n the order of the matrix. The parameter ξ considers the number of vector operations except the operations of the matrix-vector product and the ratio of the execution times of multiplication, division etc. operations and the addition operation; it is therefore dependent on the processor architecture. The numerator in (7) is proportional to the number of operations of one partial iteration on processor k, the denominator is proportional to the total number of operations of one iteration. It shall be remarked that for $\xi \to 0$ each processor gets nearly the same number of non-zeros and for $\xi \to \infty$ nearly the same number of rows. The first case means that the execution time of all vector-vector operations is neglectable compared with the execution time of the matrix-vector product. In the second case, the excution time of the matrix-vector product hardly contributes to the total execution time.

With these considerations, the contribution of the matrix-vector product to one iteration can be approximated by

$$a_{\text{MVP}} \approx \frac{e}{e + \xi n} = \frac{1}{1 + \xi / m_z}, \quad \text{for } e, n \gg 10. \tag{8}$$

$m_z = e/n$ is the mean number of non-zeros per row. Additionally, (8) provides a means for measuring ξ. If a_{MVP} is determined by timings an approximation of ξ can be computed by

$$\xi \approx m_z \left(\frac{1}{a_{\text{MVP}}} - 1\right). \tag{9}$$

On the INTEL i860XR, the timings result in an approximative value ξ of about 8.3 for the considered CG method.

The data distribution according to criterion (7) is shown in (10) by distributing the array a^w to four processors.

$$
\begin{array}{lll}
\text{Processor 0: } a_0^w & = & (1 \mid 9\ 2 \mid 10\ 9\ 3), \\
\text{Processor 1: } a_1^w & = & (10\ 4\ 18\ 14\ 12\ 11 \mid 11\ 5\ 17), \\
\text{Processor 2: } a_2^w & = & (15\ 14\ 6 \mid 12\ 17\ 7\ 15), \\
\text{Processor 3: } a_3^w & = & (18\ 8).
\end{array}
\tag{10}
$$

4.2 Communication Scheme

On a distributed memory system, the computation of the matrix-vector product requires communication because each processor owns only a partial vector. For the efficient computation of the matrix-vector product, it is necessary to develop a suitable communication scheme by preprocessing the distributed column index arrays.

First, the arrays a_k^s are analysed on each processor k to determine which data result in accesses to components of d_i of other processors. Then, a_k^s and a_k^w are reorderd in such a way that the data which result in accesses to processor h are collected in block h. The data of block h succeed row-wise one another with increasing column index per row. Block k is the first block in a_k^s and a_k^w and contains the data which result in local accesses. The goal of the reordering is performing computation and communication overlapped.

The principle of the first reordering scheme is shown in (11) for the data distribution from (10) and the matrix-vector product Ad_i from algorithm 2.1. A second reordering and communication scheme will be discussed below. Here, merely array a_1^s is analysed and reordered.

$$
\begin{array}{lll}
\text{Processor 0: } a_0^s & = & (1 \mid 3\ 2 \mid 4\ 2\ 3), \quad d_{i,0} = (d_i^1\ d_i^2\ d_i^3) \\
\underline{\text{Processor 1:}}\ a_1^s & = & (\boxed{3}\ 4\ \boxed{8}\ \boxed{6}\ \boxed{7}\ 5 \mid 4\ 5\ \boxed{7}), \quad d_{i,1} = (d_i^4\ d_i^5) \\
\text{Processor 2: } a_2^s & = & (7\ 4\ 6 \mid 4\ 5\ 7\ 6), \quad d_{i,2} = (d_i^6\ d_i^7) \\
\text{Processor 3: } a_3^s & = & (4\ 8), \quad d_{i,3} = (d_i^8)
\end{array}
\tag{11}
$$

$$
\underline{\text{Reordering:}}\ a_1^s = (\underbrace{4\ 5 \mid 4\ 5}_{1} \parallel \underbrace{\boxed{3}}_{0} \parallel \underbrace{\boxed{6}\ \boxed{7} \mid \boxed{7}}_{2} \parallel \underbrace{\boxed{8}}_{3})
$$

Computing the operation row times vector of the matrix-vector product of processor 1, the index 3 results in an access to component d_i^3 of processor 0, the index 8 to d_i^8 of processor 3, and the indices 6 and 7 in accesses to d_i^6 and d_i^7 of processor 2. The data blocks in (11) are separated by double dashes for elucidation; the blocks have been numbered below the brackets. After reordering, the data of block 1 result in local accesses, the data of block 0 in accesses to processor 0, the data of block 2 in accesses to processor 2, and the data of block 3 in accesses to processor 3.

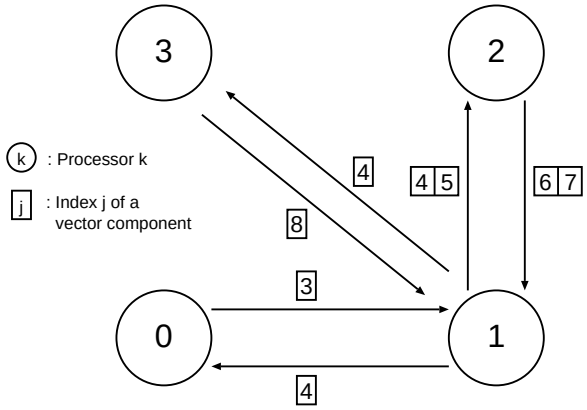

Figure 2: Communication scheme, reordering 1

After having analysed the column index array a_k^s, each processor k knows which components of d_i must be required of which processors. This information is broadcasted to all processors. Two variants have been investigated. First, the minimum and maximum index of the required components are sent; in this scope, there can be indices of components which are not needed. Second, all indices of the requisite components are sent. Thereafter, each processor can decide which data must be sent to which processors. This communication scheme is determined once before starting the parallel CG algorithm and applies unchanged to each iteration.

The communication scheme for the example discussed above is displayed in figure 2.

Processor 1 e.g. receives the third component of d_i from processor 0, the sixth and seventh component from processor 2 and the eighth component from processor 3. On the other side, the fourth component of processor 1 is sent to processor 0, the fourth and fifth to processor 2 and the fourth to processor 3.

In figure 3, the parallel computation of the matrix-vector product is described for algorithm 2.1.

First, on each processor, the data which are necessary for other processors are sent asynchronously. After having executed asynchronous receive-routines for receiving non-local data, all local computations are performed, in particular the local part of the matrix-vector product. Then each processor waits until the data of an arbitrary processor arrive and continues the computation of the matrix-vector product. Thereafter, each processor awaits the data of other processors until the computation of the matrix-vector product is complete. Computation and communication are performed overlapped. While required data are on the network, operations with local or already arrived data of other processors are executed.

In the second reordering scheme, the data blocks, built as discussed before, are sent to the processors which own the corresponding components of the vector

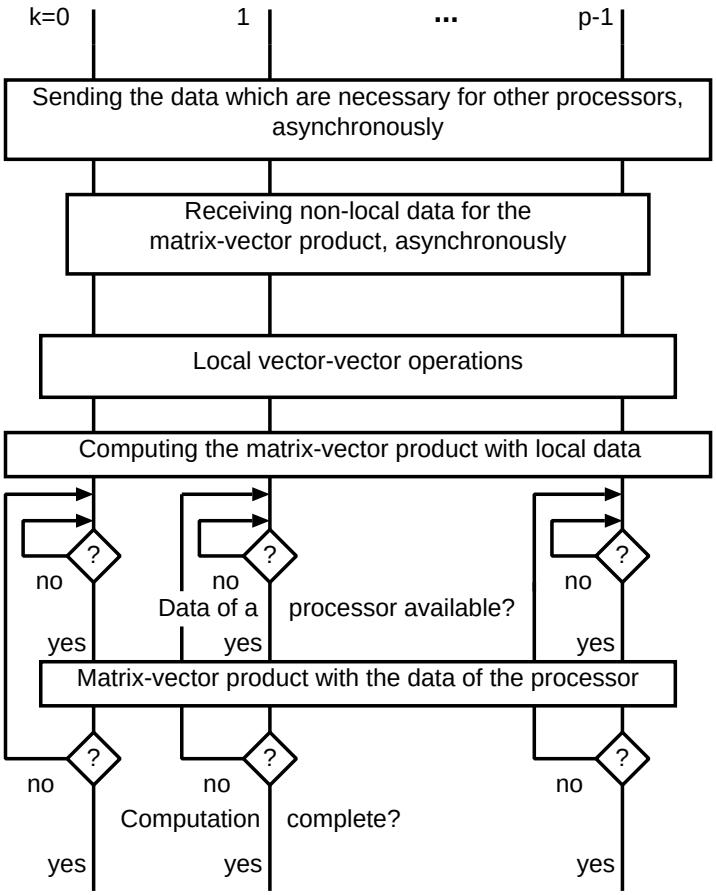

Figure 3: The parallel matrix-vector product, reordering 1

of the matrix-vector product. The goal is to increase the number of local computations while required data are on the network. In this case, the processors compute partial results of the result vector of the matrix-vector product. Then, $y_{k,l}$ denotes the partial result of $y_k = A_k d_i$ of processor k which is computed on processor l. After the computation, the partial results except the local one are sent to the corresponding processors and then are added to the local result of the receiving processor.

The new distribution of the matrix data is presented in (12).

The first number of the blocks in (12) denotes the processor to which the partial result is sent; the second number indicates the processor on which the computation is performed. Processor 1 e.g. computes the local result $y_{1,1}$ with the first block, the partial result $y_{0,1}$ of processor 0 with the second block, the partial result $y_{2,1}$ of processor 2 with the third block, and the partial result $y_{3,1}$ of processor 3 with the fourth block, respectively.

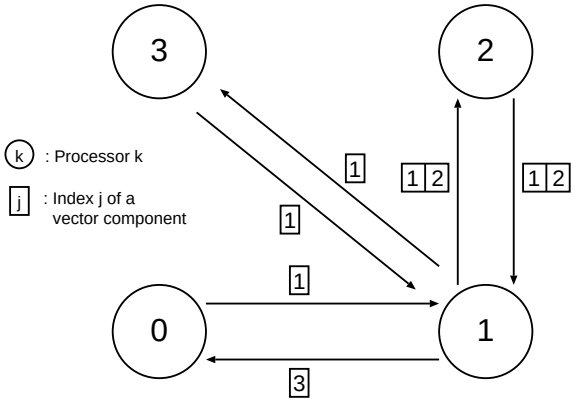

Figure 4: Communication scheme, reordering 2

$$
\begin{aligned}
\text{Processor 0: } a_0^s &= (\underbrace{1 \mid 2\ 3 \mid 2\ 3}_{0,0} \parallel \underbrace{\boxed{3}}_{1,0}), \quad d_{i,0} = (d_i^1\ d_i^2\ d_i^3) \\
\text{Processor 1: } a_1^s &= (\underbrace{4\ 5 \mid 4\ 5}_{1,1} \parallel \underbrace{\boxed{4}}_{0,1} \parallel \underbrace{\boxed{4} \mid \boxed{4}\ \boxed{5}}_{2,1} \parallel \underbrace{\boxed{4}}_{3,1}), \quad d_{i,1} = (d_i^4\ d_i^5) \\
\text{Processor 2: } a_2^s &= (\underbrace{6\ 7 \mid 6\ 7}_{2,2} \parallel \underbrace{\boxed{6}\ \boxed{7} \mid \boxed{7}}_{1,2}), \quad d_{i,2} = (d_i^6\ d_i^7) \qquad (12) \\
\text{Processor 3: } a_3^s &= (\underbrace{8}_{3,3} \parallel \underbrace{\boxed{8}}_{1,3}), \quad d_{i,3} = (d_i^8)
\end{aligned}
$$

Figure 4 shows the communication scheme for the block distribution from (12).

Processor 1 e.g. sends a value to processor 0 which must be added to the third component of y_0. On the other side, processor 1 receives a value from processor 0 which must be added to the first component of y_1.

In figure 5, the parallel computation of the matrix-vector product is presented for the second reordering and communication scheme.

First, asynchronous receive-routines for receiving all neccessary partial results of other processors are executed on each processor. After that, each processor computes the partial results which are sent to other processors. The computation is performed per data block; the results are asynchronously sent to the corresponding processors after each computation. Then, all local computations are performed, in particular the computation of the local part of the matrix-vector product. Thereafter, each processor waits until the data of an arbitrary processor arrive and then adds the values to the corresponding components of the local result. This is repeated until the computation of the matrix-vector product is complete. Computation and communication are performed overlapped.

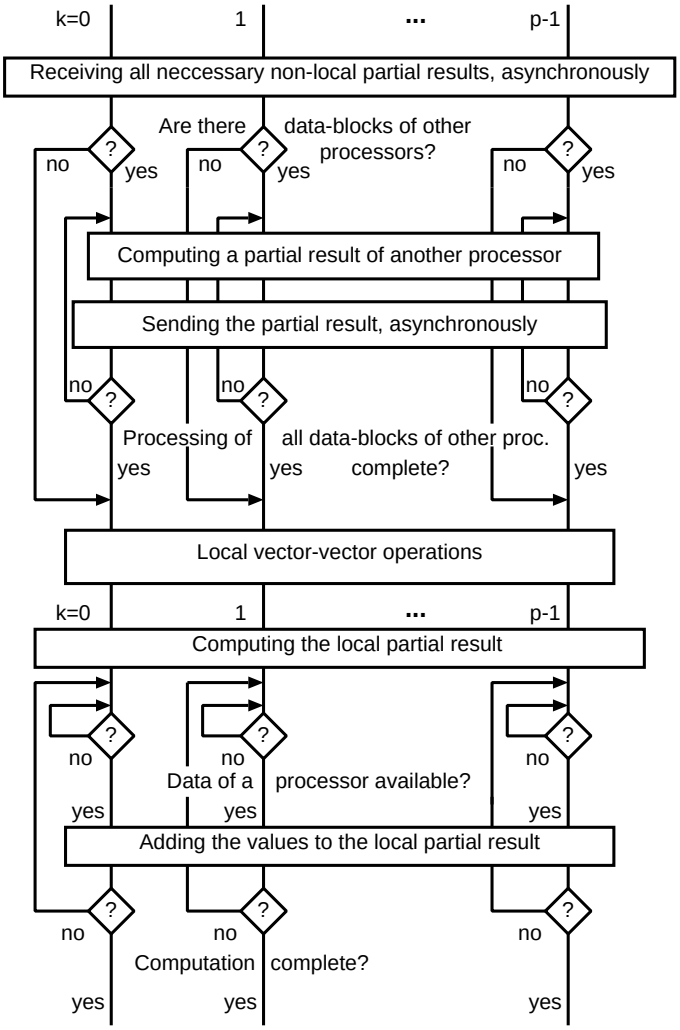

Figure 5: The parallel matrix-vector product, reordering 2

Since partial results of the matrix-vector product are exchanged most computations are local. Merely the summation of the partial results follows after the receiving of non-local data. The disadvantage of this method is that load balancing is not guaranteed any more after the new distribution of the blocks; some processors can own more or larger data blocks than other ones. However, this scheme allows arbitrary data distributions; each processor can get arbitrary parts of arbitray rows which need not succeed one another. For a specific FE application, suitable data distributions for this scheme can be found considering the discretization mesh. The data distribution and the communication scheme presented here do not require any knowledge about a specific discretization mesh; the schemes are determined automatically by analysing the column indices of the non-zero matrix elements.

5 Results

The numerical and performance tests of the developed parallel CG methods have been performed on the distributed-memory system iPSC/860 of the Research Centre Jülich. The INTEL computer system has 32 processors with 16 Megabyte private memory each interconnected by a hypercube-network. The maximum transfer rate is 2.8 Megabyte/second per channel in both directions.

5.1 Numerical Results

The tests presented here have been carried out with one equation system each of the FE models from environmental science and structural mechanics. In table 1, numerical data of the coefficient matrices and for the convergence of the CG method are indicated.

	Environmental science	Structural mechanics
Rows	49392	25222
Non-zeros	1242814	3856386
Density	0.05%	0.6%
Non-zeros per row, max.	27	485
m_z	25.2	152.9
a_{MVP}	75%	95%
CG method: max. scal. abs. diff. $\leq 10^{-5}$		
Iterations without scaling	390	1444
Iterations with scaling	84	658
$\|g_{i+1}\|_2$	4.5×10^{-4}	1.5×10^{-5}

Table 1: Numerical data of the considered large sparse matrices

The matrix from environmental science has 49392 rows, that from structural mechanics 25222. In the first case, the mean number of non-zeros per row is near the maximum number. This is caused by a regular discretization mesh. For the second case, the mean and the maximum number are considerably different; the discretization mesh is much more irregular. The operational contribution of the matrix-vector product to one iteration is 75% for the matrix from environmental science and 95% for the matrix from structural mechanics.

Below in table 1, the number of CG iterations with and without diagonal scaling is given. The iteration has been stopped when the maximum scaled absolute difference from (1) has been less than or equal to 10^{-5}; this corresponds to a precision of the solution vector of about five decimals. With diagonal scaling, the number of iterations is considerably smaller in both cases. The contribution

of this preconditioner to the total execution time is in both cases below 1%. For the preconditioned method, the euclidean norm of the residue after 84 and 658 iterations, respectively, is given in addition.

The sparsity patterns of both matrices are shown in the figures 6 and 7, respectively.

The matrix from environmental science has essentially band structure with a maximum bandwidth of 2375. The matrix from structural mechanics has a much more irregular structure; the maximum bandwidth is 3474.

By reducing the bandwidth of the matrices, the communication overhead in each iteration of the CG method can be decreased. Since communication is necessary for the operation row times vector of the matrix-vector product, a smaller bandwidth results in smaller message length or even in communication with fewer processors. Here, the matrix is reordered by the reverse Cuthill-McKee (RCM) scheme [16]. In FE models, this scheme is frequently used for the assembly of the coefficient matrix; it is performed merely once if the mesh does not change, whereas in many cases equation systems are frequently solved, e.g. in each time step of a time dependent problem or in each iterative step of a nonlinear problem which is solved by linearization.

In the figures 8 and 9, the sparsity patterns of both matrices with bandwidth reduction are presented.

For the matrix from environmental science, the bandwidth is reduced by 45%; the maximum bandwidth is 1303. The maximum bandwidth of the matrix from structural mechanics after applying the reverse Cuthill-McKee scheme is 2989; this is a reduction by merely 14%.

5.2 Performance Results

In the first four investigations, bandwidth reduction has not been applied to the matrices.

Figure 10 shows execution times per iteration of the parallel CG method for both the presented storage schemes using 16 processors. In this investigation, each processor has got nealy the same number of succeeding rows.

The execution times in figure 10 hardly differ for the storage schemes in 2-dimensional and 1-dimensional arrays. The times for the scheme in 1-dimensional arrays are slightly increased since in this case a third array is neccessary for addressing; the costs for accessing the matrix elements are higher compared with the scheme in 2-dimensional arrays. The storage requirement, however, is about 147 Megabyte for the matrix from structural mechanics using the storage scheme in 2-dimensional arrays and merely 47 Megabyte using the storage scheme in 1-dimensional arrays. This is caused by a very different number of non-zeros per row of this matrix. For the matrix from environmental science, the storage requirement is 17 Megabyte in the first and 16 Megabyte in the second case. The number of non-zeros per row is the same for most rows. Because of the less

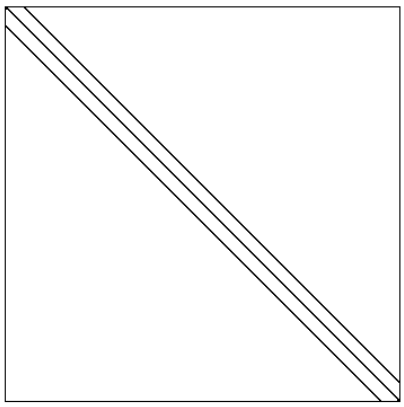

Figure 6: Sparsity pattern of the matrix from environmental science

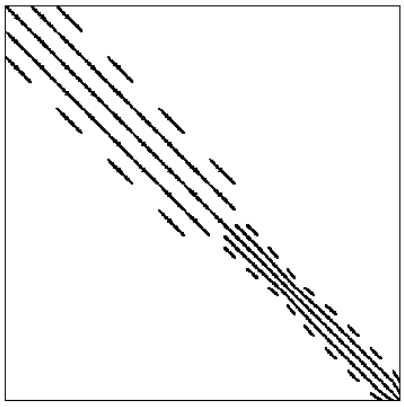

Figure 7: Sparsity pattern of the matrix from structural mechanics

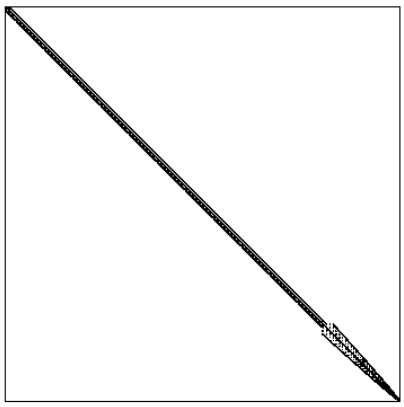

Figure 8: Sparsity pattern of the matrix from environmental science with bandwidth recuction

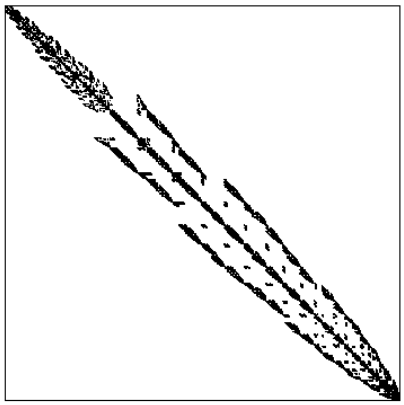

Figure 9: Sparsity pattern of the matrix from structural mechanics with bandwidth recuction

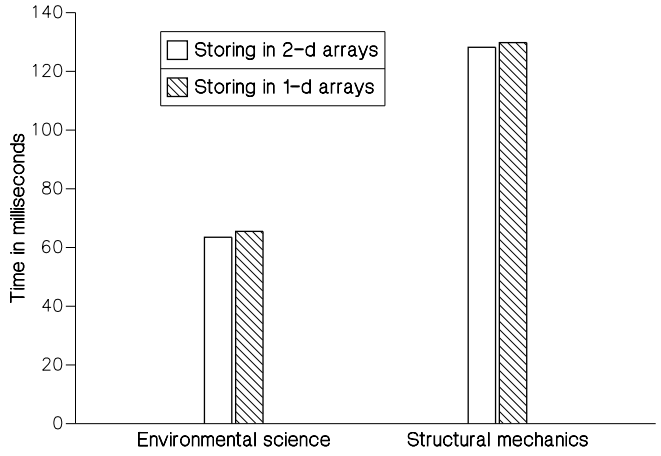

Figure 10: Execution times per iteration, different storage schemes, 16 processors

storage requirements, the storage scheme in 1-dimensional arrays is applied in all following measurements.

In figure 11, execution times per iteration on 32 processors are presented for the three considered data distributions.

For the matrix from environmental science, the execution times are nearly the same because the matrix has quite a regular structure. For the matrix from structural mechanics, the execution times using the criteria "same number of non-zeros" and "same number of operations" are reduced by 19% compared with the time using the criterion "same number of rows". Because of the very different number of non-zeros per row, the operations for the computation of the matrix-vector product are not uniformly distributed to each processor applying the latter criterion. The times for the criteria "same number of non-zeros" and "same number of operations" are nearly the same since the contribution of the time for computing the matrix-vector-product to the total time for one iteration is 95%. In all following investigations, the criterion "same number of operations" is applied.

Figure 12 shows execution times per iteration on 32 processors for different communication schemes.

In the first case, the vector components of the vector of the matrix-vector product from the minimum to the maximum index which have been determined by analysing the column index array are sent. In the second case, only the components which are necessary for the computation are delivered as packed list, and in the third case, the components of partial results of the matrix-vector product are sent, also as packed list. For the matrix from environmental science, using

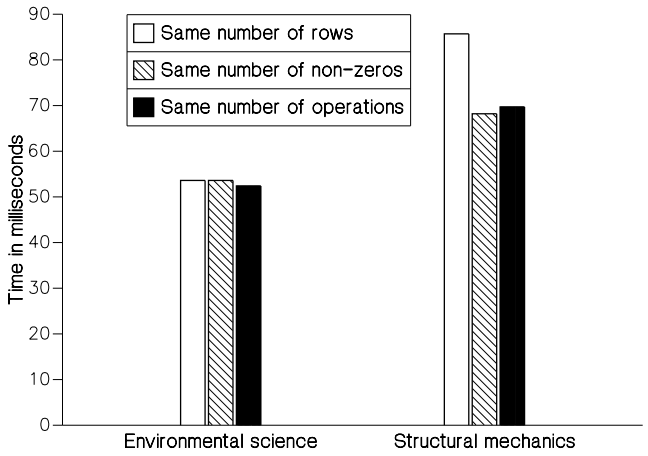

Figure 11: Execution times per iteration, different data distributions, 32 processors

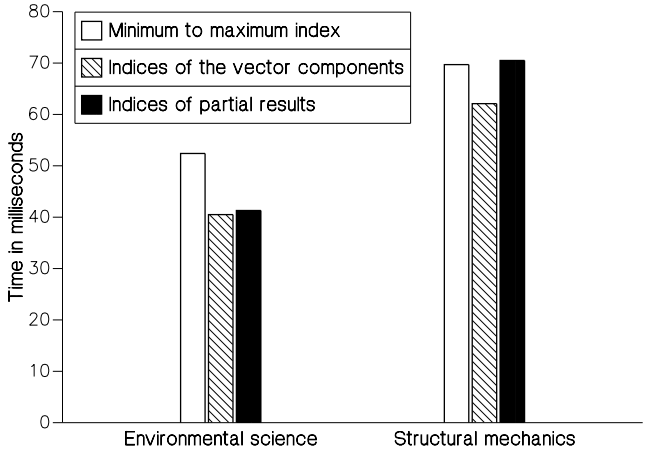

Figure 12: Execution times per iteration, different communication schemes, 32 processors

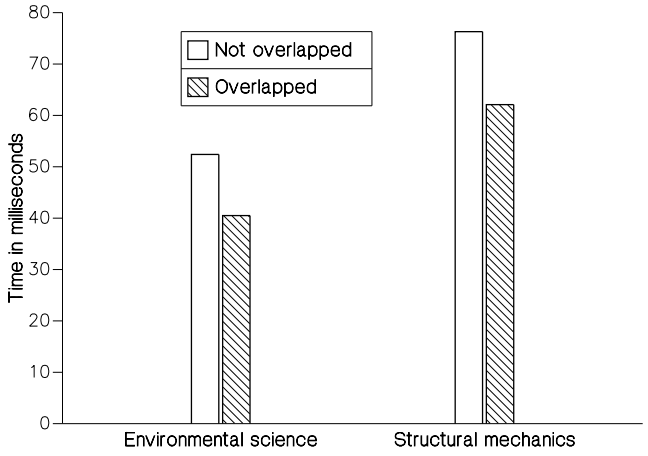

Figure 13: Execution times per iteration, CG method, 32 processors

case 2 reduces the time by about 23% compared with case 1 and by about 9% for the matrix from structural mechanics. This indicates that in case 1 considerably more components than necessary are sent. Case 3 does not result in further improvement. On the contrary, the time increases for the matrix from structural mechanics. Because of the irregular structure of this matrix, the new distribution of the data blocks destroys the load balancing. In the following investigations, the communication scheme according to case 2 is applied since the best results have been achieved with this scheme for the considered matrices.

In figure 13, execution times per iteration on 32 processors with and without communication and computation performed overlapped are presented. The overlapped execution reduces the execution times by nearly 20%.

In figure 14, speedups on 4 to 32 processors are shown with and without bandwidth reduction of the matrices. The equation system from environmental science together with the program code and the remaining data requires the memory of more than two processors, that from structural mechanics the memory of more than four processors. For up to four and, in the second case, up to eight processors, linear speedup was assumed because nearly linear speedup was observed in tests with smaller systems of equations for up to 8 processors.

For 16 processors and without bandwidth reduction, the speedup is 13.2 in the first case and 15.2 in the second case. This corresponds to efficiencies of 83% and 95%. With bandwidth reduction, the speedups increase to 14.6 and 15.6, respectively; the efficiecies are 91% and 97%. For 32 processors, speedups of 21.6 and 27.2 without bandwidth reduction or of 24.8 and 28.5 with bandwidth reduction are achieved. The efficiencies decrease to 68% and 85% without bandwidth

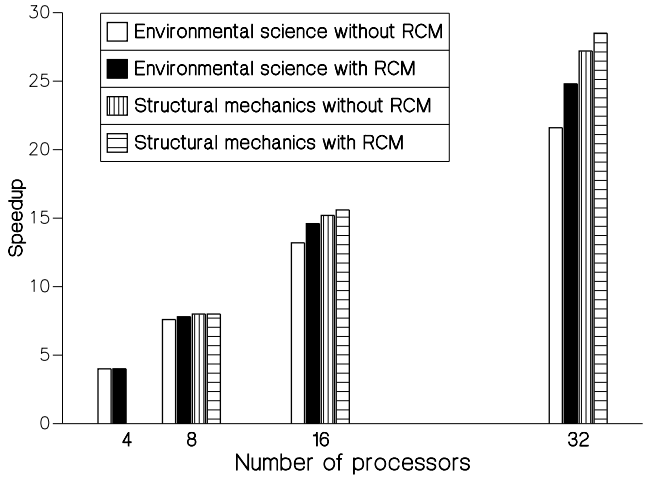

Figure 14: Speedups, CG method

reduction or 78% and 89% with bandwidth reduction because the communication overhead increases.

6 Conclusions

The developed parallel CG algorithms have been shown to be well suited for the considered large sparse matrices from FE models; they are employed in both the projects from environmental science and structural mechanics. On a distributed memory system, the data distribution and communication scheme determined by preprocessing together with the overlapped execution of communication and local computations result in flexible algorithms. These algorithms perform well for large sparse matrices of very different sparsity patterns.

Bandwidth reduction decreases the communication overhead additionally; applying reordering schemes like the reverse Cuthill-McKee method is recommended for FE models.

In current investigations, other data distributions and other storage schemes for sparse matrices and their influence on the communication scheme are tested as well as polynomial preconditioning [3] [13] as convergence accelerator. This preconditioner allows using the same parallelization strategies as described for the CG algorithm in this report. Furthermore, the applicability of these principles to the QMR algorithm for solving non-hermitian systems of linear equations [9] will be investigated.

Moreover, further research is scheduled to parallelize the Lanczos algorithm

for solving large sparse symmetric eigenproblems, which has a deep theoretical connection to the CG method [6]. The Lanczos algorithm requires essentially the computation of matrix-vector products and the determination of eigenvalues and eigenvectors of symmetric tridiagonal matrices. A parallel solver for determining all eigenvalues of large real symmetric tridiagonal matrices has already been developed [5]. For determining the eigenvectors of the tridiagonal matrices, inverse iteration is a suitable method and can be done perfectly in parallel if the corresponding eigenvalues are known.

References

[1] 3DFEMWATER: a three-dimensional finite element model of water flow through saturated-unsaturated media. Oak Ridge National Laboratory. *ORNL-6386*, 1987

[2] SMART, Benutzerhandbücher. Institut für Statik und Dynamik der Luft- und Raumfahrtkonstruktionen der Universität Stuttgart. *ISD-Berichte*, 1976-1992.

[3] S.F. Ashby. Minimax polynomial preconditioning for hermitian linear systems. *SIAM J. Matrix Anal. Appl.*, 12:766–789, 1991.

[4] C. Aykanat, F. Özgüner, D.S. Scott. Vectorization and parallelization of the conjugate gradient algorithm on hypercube-connected vector processors. *Microprocessing and Microprogramming*, 29:67–82, 1990.

[5] A. Basermann, P. Weidner. A parallel algorithm for determining all eigenvalues of large real symmetric tridiagonal matrices. *Parallel Computing*, 18:1129–1141, 1992.

[6] J.K. Cullum, R.A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations.* Volume I: Theory, Birkhäuser, Boston Basel Stuttgart, 1985.

[7] N. Feistl. *Das Verfahren der konjugierten Gradienten auf Vektorrechern.* Diplomarbeit, Ludwig-Maximilians-Universität, München, Juli 1990.

[8] P. Fernandes, P. Girdinio. A new storage scheme for an efficient implementation of the sparse matrix-vector product. *Parallel Computing*, 12:327–333, 1989.

[9] R.W. Freund, N.M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60:315–339, 1991.

[10] M.R. Hestenes, E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[11] C.P. Kruskal, L. Rudolph, M. Snir. Techniques for parallel manipulation of sparse matrices. *Theoretical Computer Science*, 64:135–157, 1989.

[12] O.A. McBryan, E.F. Van de Velde. Matrix and vector operations on hypercube parallel processors. *Parallel Computing*, 5:117–125, 1987.

[13] J.M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems.* Plenum Press, New York London, 1988.

[14] S. Pissanetsky. *Sparse Matrix Technology.* Academic Press, London Orlando, 1984.

[15] U. Schendel. *Sparse-Matrizen.* R. Oldenbourg Verlag, München Wien, 1. Auflage, 1977.

[16] H.R. Schwarz. *FORTRAN-Programme zur Methode der finiten Elemente.* B. G. Teubner, Stuttgart, 1981.

[17] H. Vereecken, G. Lindenmayr, A.Kuhr, D.H. Welte, A. Basermann. Numerical modelling of field scale transport in heterogeneous variably saturated porous media. *KFA/ICG-4 Internal Report No. 500393*, January 1993.