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

Interner Bericht

Preconditioned CG methods for sparse matrices on massively parallel machines

Achim Basermann, Björn Reichel, Christof Schelthoff

KFA-ZAM-IB-9531

Dezember 1995
(Stand 19.12.95)

Dieser Bericht wurde zur Publikation eingereicht.

Preconditioned CG methods for sparse matrices on massively parallel machines

A. Basermann, B. Reichel, and C. Schelthoff

Central Institute for Applied Mathematics, Research Centre Jülich GmbH, 52425 Jülich, Germany

Abstract

Conjugate gradient (*CG*) methods to solve sparse systems of linear equations play an important role in numerical methods for solving discretized partial differential equations. The large size and the condition of many technical or physical applications in this area result in the need for efficient parallelization and preconditioning techniques of the CG method, in particular on massively parallel machines. Here, the data distribution and the communication scheme for the sparse matrix operations of the preconditioned CG are based on the analysis of the indices of the non-zero elements. Polynomial preconditioning is shown to reduce global synchronizations considerably, and a fully local incomplete Cholesky preconditioner is presented. On a PARAGON XP/S 10 with 138 processors, the developed parallel methods outperform diagonally scaled CG markedly with respect to both scaling behavior and execution time for many matrices from real finite element applications.

Key words. Conjugate gradient method; polynomial preconditioning; incomplete Cholesky preconditioning; sparse matrices; massively parallel machine; distributed memory; data distribution; communication scheme; reordering.

1 Introduction

For the analysis and solution of discretized ordinary or partial differential equations it is necessary to solve systems of equations with coefficient matrices of different sparsity patterns, depending on the discretization method. In many cases, the use of the finite element method (*FE*) results in largely unstructured systems of equations.

A frequently used algorithm to solve such systems with symmetric positive definite coefficient matrix is the method of conjugate gradients (*CG*). Since the underlying problems often result in very ill-conditioned matrices, preconditioning techniques are necessary to obtain both acceptable convergence and accuracy of CG. In this article, we investigate variants of polynomial and incomplete Cholesky preconditioners that markedly reduce the iterations of the simply diagonally scaled CG and are shown to be well suited for massively parallel machines.

The basic operations of the pure CG iteration as well as the polynomially preconditioned method [1] are matrix-vector products with the coefficient matrix and vector-vector computations. For incomplete Cholesky preconditioning, the factorization of the matrix before the CG iteration and a forward/back-substitution per iteration [9] are required in addition.

To parallelize these operations on a multiprocessor system with distributed memory, we present a data distribution and a communication scheme based on the analysis of the non-zero matrix elements. By a suitable block-reordering of the matrix, the overlapped execution of computation and communication is supported to reduce waiting times. Factorization and forward/back-substitution of the developed incomplete Cholesky preconditioner are only performed on local blocks so that these

operations do not require 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 they both are exploited in each iteration. The schemes can be reused as long as the sparsity pattern of the matrix, which is determined by the discretization mesh and the element types, does not change. For example, they can be used in each time step of a time dependent problem or in each iterative step of a nonlinear problem that is solved by linearization.

On massively parallel systems, the computation of inner products in the CG iteration may be very costly since it needs global communication. To save synchronization points, inner products are grouped so that each iteration of the developed algorithms only requires one global communication. In addition, the contribution of inner products to the costs per iteration is considerably reduced by the preconditioning techniques used.

Performance tests of the algorithms were carried out on the distributed memory system PARAGON XP/S 10 of the Research Centre Jülich with sparse matrices from various FE applications.

2 Theoretical background

2.1 The conjugate gradient method

The method of conjugate gradients — originally developed in 1952 by Hestenes and Stiefel [10] — is an algorithm for solving systems of linear equations $\boldsymbol{A}x = b$, particularly when $\boldsymbol{A}$ is a sparse coefficient matrix. The method applies to symmetric positive definite (*spd*) matrices $\boldsymbol{A} \in \mathbb{R}^{n \times n}$.

Aykanat e.a. [3] as well as Chronopoulos and Gear [7] suggested a modified CG iteration (Algorithm 2.1) that has better parallelization properties than the original method.

Algorithm 2.1. The modified CG method

Init.: $x_0,\ r_0 = b - \boldsymbol{A}x_0,\ d_0 = r_0$

$i = 0, 1, \ldots$

$$
\begin{aligned}
y_i &= \boldsymbol{A}d_i \\
\delta_i &= r_i^T r_i \\
\alpha_i &= \frac{\delta_i}{d_i^T y_i} \\
\beta_i &= \alpha_i \frac{y_i^T y_i}{d_i^T y_i} - 1 \\
x_{i+1} &= x_i + \alpha_i d_i \\
r_{i+1} &= r_i - \alpha_i y_i \\
d_{i+1} &= r_{i+1} + \beta_i d_i
\end{aligned}
$$

until convergence.

The main difference between the original and the modified algorithm is that in the modified version all dot products are computed without any operations in between. Therefore, 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 to determine the global values.

The computation of the dot product $r_i^T r_i$ can be avoided using the value of $\beta_{i-1} r_{i-1}^T r_{i-1}$ from the previous iteration, but with this the iteration has been shown to be less robust than the original

method for some test matrices. To obtain the same robustness as the original CG, we compute $r_i^T r_i$ in each iteration. This results in an additional dot product at the beginning of each iteration, but does not affect the more advantageous parallelization properties of Algorithm 2.1 compared with the original method.

The convergence of the CG method depends on the eigenvalue distribution of $\boldsymbol{A}$ [11]. A criterion for the width of the spectrum is the euclidean condition number that is for spd matrices

$$\kappa := \frac{\lambda_{max}(\boldsymbol{A})}{\lambda_{min}(\boldsymbol{A})} \quad (\geq 1)\,.$$

With $\gamma := \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$, the distance to the exact solution x^* in the i-th iteration is bounded by

$$\| x_i - x^* \|_2 \leq 2\sqrt{\kappa}\gamma^i \| x_0 - x^* \|_2 \ .$$

The right hand side increases with growing condition number. Hence lower condition numbers usually accelerate convergence.

A simple but often very efficient preconditioner is diagonal scaling [5, 6, 11, 12]. The CG method is applied to the scaled system $\boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{D}^{-1}\tilde{x} = \boldsymbol{D}^{-1}b$ with $d_{j,j} = \sqrt{a_{j,j}},\ j = 1, \ldots, n$, as the elements of the diagonal matrix $\boldsymbol{D}$ and $\tilde{x} = \boldsymbol{D}x$. The original solution is given by $x = \boldsymbol{D}^{-1}\tilde{x}$.

2.2 Polynomial preconditioning

The basic idea of polynomial preconditioning to accelerate the convergence of CG is to find a matrix polynomial $\boldsymbol{C}$ — the preconditioner — to perform the transformation $\boldsymbol{C}\boldsymbol{A}x = \boldsymbol{C}b$ with $\kappa(\boldsymbol{C}\boldsymbol{A}) \ll \kappa(\boldsymbol{A})$. Of course, we do not evaluate the matrix product $\boldsymbol{C}\boldsymbol{A}$ explicitly, but we replace the matrix-vector product $\boldsymbol{A}d_i$ by $\boldsymbol{C}(\boldsymbol{A}d_i)$ in the CG iteration. That means performing additional matrix-vector products. Furthermore, b is replaced by $\boldsymbol{C}b$.

Here, we choose $\boldsymbol{C}$ as a polynomial of degree m in $\boldsymbol{A}$ that is $\boldsymbol{C} = \mathcal{C}(\boldsymbol{A}) = \sum_{j=0}^{m} \nu_j \boldsymbol{A}^j$. Thus the new system to solve is

$$\underbrace{\mathcal{C}(\boldsymbol{A})\boldsymbol{A}}_{\mathcal{P}(\boldsymbol{A})} x \quad = \quad \mathcal{C}(\boldsymbol{A})b\,.$$

$\mathcal{P}(\boldsymbol{A})$ is a polynomial of degree $k = m + 1$, and we obtain the CG algorithm with polynomial preconditioning by replacing the initial residual in Algorithm 2.1 by $r_0 = \mathcal{C}(\boldsymbol{A})(b - \boldsymbol{A}x_0)$ and $y_i = \boldsymbol{A}d_i$ by $y_i = \mathcal{P}(\boldsymbol{A})d_i$.

To construct $\mathcal{C}(\boldsymbol{A})$, we apply scaled and translated Chebyshev polynomials [14] that require estimations $\tilde{\lambda}_{min}$ and $\tilde{\lambda}_{max}$ for the smallest and the largest eigenvalue of $\boldsymbol{A}$. The connection between the Lanczos algorithm for symmetric eigenproblems and the CG method allows to improve these eigenvalues during the iteration, and so the algorithm becomes adaptive [1].

By using polynomial preconditioning, the reduction of the iterations is bounded by the factor k. Since we have to calculate k matrix-vector products in each step of the preconditioned CG rather than one in the basic CG, the total number of matrix-vector products usually increases. However, we only need one synchronization step per iteration in the basic as well as in the polynomially preconditioned method. Hence the reduction of iteration steps leads to markedly less global synchronization. A further increase of the polynomial's degree enlarges the number of matrix-vector products, but decreases the total number of global synchronizations [14].

Finally we should remark that calculating the coefficients of the polynomial is not necessary. The *Chebyshev iteration* [8], a 3-term recursion, can be exploited. Algorithm 2.2 displays the iteration

for the calculation of $\mathcal{C}(\boldsymbol{A})z$ where $\mathcal{C}(\boldsymbol{A})$ is a polynomial of degree m. The value of $\mathcal{P}(\boldsymbol{A})d_i$ from the preconditioned CG iteration is computed by $\boldsymbol{A}(\mathcal{C}(\boldsymbol{A})d_i)$.

Algorithm 2.2. Chebyshev iteration to calculate $w = \mathcal{C}(\boldsymbol{A})z$

Init.: $\eta = \frac{\tilde{\lambda}_{min}+\tilde{\lambda}_{max}}{2}$, $\vartheta = \frac{\tilde{\lambda}_{max}-\tilde{\lambda}_{min}}{2}$, $\sigma = \frac{\vartheta}{\eta}$, $w_0 = 0$, $w_1 = \frac{1}{\eta}z$, $\rho_1 = 2$
$i = 2, \ldots, m+1$

$$\begin{aligned} \rho_i &= \frac{4}{4-\sigma^2\rho_{i-1}} \\ v_{i-1} &= z - \boldsymbol{A}w_{i-1} \\ w_i &= \rho_i(w_{i-1} - w_{i-2} + \frac{1}{\eta}\, v_{i-1}) + w_{i-2} \end{aligned}$$

$w = w_{m+1}$.

2.3 Incomplete Cholesky preconditioning

Incomplete Cholesky factorization methods have been shown to be very efficient preconditioners for structural engineering problems [2] [9]. Axelsson [2] proves the existence of incomplete factorizations for M-matrices, positive definite matrices on block tridiagonal form, and block H-matrices. We describe a simple but efficient method to determine an incomplete Cholesky factorization that exists for spd matrices of arbitrary sparsity pattern. This method is highly parallel both in the factorization part and in solving the preconditioning system.

Algorithm 2.3 displays a formulation of the preconditioned CG iteration (*PCG*) that is suited for applying the incomplete Cholesky factorization; Algorithm 2.4 is a modified variant with better parallel properties.

Algorithm 2.3. PCG

Init.: x_0, $r_0 = b - \boldsymbol{A}x_0$, $\boldsymbol{C}z_0 = r_0$, $d_0 = z_0$
$i = 0, 1, \ldots$

$$\begin{aligned} y_i &= \boldsymbol{A}d_i \\ \alpha_i &= \frac{r_i^T z_i}{d_i^T y_i} \\ x_{i+1} &= x_i + \alpha_i d_i \\ r_{i+1} &= r_i - \alpha_i y_i \\ \boldsymbol{C}z_{i+1} &= r_{i+1} \\ \beta_i &= \frac{r_{i+1}^T z_{i+1}}{r_i^T z_i} \\ d_{i+1} &= z_{i+1} + \beta_i d_i \end{aligned}$$

until convergence.

Algorithm 2.4. Modified PCG

$i = 0, 1, \ldots$

$$\begin{aligned} y_i &= \boldsymbol{A}d_i \\ \boldsymbol{C}v_i &= y_i \\ \alpha_i &= \frac{r_i^T z_i}{d_i^T y_i} \\ \beta_i &= 1 - \frac{2\alpha_i z_i^T y_i - \alpha_i^2 y_i^T v_i}{r_i^T z_i} \\ z_{i+1} &= z_i - \alpha_i v_i \\ x_{i+1} &= x_i + \alpha_i d_i \\ r_{i+1} &= r_i - \alpha_i y_i \\ d_{i+1} &= z_{i+1} + \beta_i d_i \end{aligned}$$

until convergence.

The preconditioner $\boldsymbol{C}$ must be a good approximation of the original matrix $\boldsymbol{A}$, and the preconditioning system $\boldsymbol{C}v = y$ must be easy to solve.

For incomplete Cholesky preconditioning, $\boldsymbol{A}$ is approximately factorized by a product of two

triangular matrices,

$$\boldsymbol{C} = \boldsymbol{L}\boldsymbol{L}^T \approx \boldsymbol{A},$$

where $\boldsymbol{L}$ is a lower triangular matrix. We get the result of the system $\boldsymbol{L}\boldsymbol{L}^T v = y$ by the successive solution of the two systems $\boldsymbol{L}w = y$ and $\boldsymbol{L}^T v = w$. Thus, the solution step of the preconditioning system is a simple forward/back-substitution.

Inner products in the preconditioned CG iteration from Algorithm 2.3 are not grouped. Therefore, if the method is parallelized on a distributed memory system, each iteration at least requires two gobal communications for synchronization. To group the inner products as in Algorithm 2.1, we transform the calcution of β_i from Algorithm 2.3 into

$$\begin{aligned}
\beta_i &= \frac{r_{i+1}^T z_{i+1}}{r_i^T z_i} = \frac{(r_i - \alpha_i y_i)^T \boldsymbol{C}^{-1}(r_i - \alpha_i y_i)}{r_i^T z_i} \\
&= \frac{r_i^T z_i - 2\alpha_i z_i^T y_i + \alpha_i^2 y_i^T \boldsymbol{C}^{-1} y_i}{r_i^T z_i} \\
&= 1 - \frac{2\alpha_i z_i^T y_i - \alpha_i^2 y_i^T v_i}{r_i^T z_i}
\end{aligned}$$

with $\boldsymbol{C}v_i = y_i$. Thus we obtain the iteration displayed in Algorithm 2.4.

Different from the iteration in Algorithm 2.3, all inner products of the new variant are computed in direct succession. Therefore one synchronization point is sufficient. The extra costs are storing an additional vector v_i and determining two additional dot products $z_i^T y_i$ and $y_i^T v_i$.

An incomplete Cholesky factorization of a matrix $\boldsymbol{A}$ with arbitrary structure, where e.g. fill-in is totally suppressed, always exists if $\boldsymbol{A}$ is spd and has M-matrix property. However, for the existence of the complete factorization the spd property is sufficient. Therefore we consider small quadratic diagonal blocks of the original matrix $\boldsymbol{A}$ and determine a complete factorization of them. These blocks are spd if $\boldsymbol{A}$ is. For the whole matrix, this means an incomplete factorization where fill-in only takes place within small quadratic diagonal blocks.

3 Storage scheme

Efficient 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 [4]. In this article, we apply the CRS format (*compressed row storage*). This format is often used in FE programs and is suited for matrices with regular as well as irregular structure. The principle of the scheme is illustrated in Fig. 1 for a matrix $\boldsymbol{A}$ with non-zeros $a_{i,j}$.

$$A = \begin{pmatrix}
\boldsymbol{a}_{1,1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \boldsymbol{a}_{2,2} & \boldsymbol{a}_{2,3} & 0 & 0 & 0 & 0 & 0 \\
0 & \boldsymbol{a}_{3,2} & \boldsymbol{a}_{3,3} & \boldsymbol{a}_{3,4} & 0 & 0 & 0 & 0 \\
0 & 0 & \boldsymbol{a}_{4,3} & \boldsymbol{a}_{4,4} & \boldsymbol{a}_{4,5} & \boldsymbol{a}_{4,6} & \boldsymbol{a}_{4,7} & \boldsymbol{a}_{4,8} \\
0 & 0 & 0 & \boldsymbol{a}_{5,4} & \boldsymbol{a}_{5,5} & 0 & \boldsymbol{a}_{5,7} & 0 \\
0 & 0 & 0 & \boldsymbol{a}_{6,4} & 0 & \boldsymbol{a}_{6,6} & \boldsymbol{a}_{6,7} & 0 \\
0 & 0 & 0 & \boldsymbol{a}_{7,4} & \boldsymbol{a}_{7,5} & \boldsymbol{a}_{7,6} & \boldsymbol{a}_{7,7} & 0 \\
0 & 0 & 0 & \boldsymbol{a}_{8,4} & 0 & 0 & 0 & \boldsymbol{a}_{8,8}
\end{pmatrix}$$

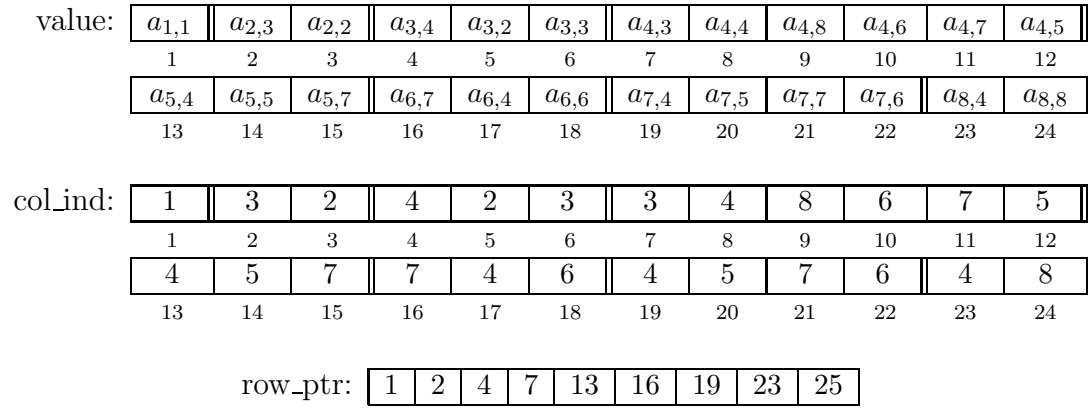

Figure 1: CRS storage scheme

The non-zeros of matrix $\boldsymbol{A}$ are stored row-wise in three one-dimensional arrays. `value` contains the values of the non-zeros, `col_ind` the corresponding column indices. The elements of `row_ptr` point to the position of the beginning of each row in `value` and `col_ind`.

The diagonal blocks for our variant of the incomplete Cholesky preconditioner (see 2.3) are stored in SKS format (*Skyline Storage*) [4] that is suited for a complete factorization of the blocks.

4 Parallelization

For the parallelization of the preconditioned CG iteration on a distributed memory system, it is essential to distribute the matrix and vector arrays suitably to each processor and to determine an efficient communication scheme for matrix-vector products depending on the sparsity pattern of the matrix.

4.1 Data distribution

In the data distribution scheme considered below, the matrix arrays `value` and `col_ind` 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. In the following, n_k and e_k denote the number of rows and non-zeros of processor k, $k = 0, \ldots, p-1$; e and n are the total numbers. g_k is the index of the first row of processor k, and z_i is the number of non-zeros of row i. For these quantities, the following equations hold: $n = \sum_{k=0}^{p-1} n_k$, $e = \sum_{k=0}^{p-1} e_k$, $g_k = 1 + \sum_{i=0}^{k-1} n_i$, and $e_k(g_k, n_k) = \sum_{i=g_k}^{g_k+n_k-1} z_i$.

In each iteration of an iterative method, the number of operations for s matrix-vector multiplications is proportional to $s\,e$; the number of operations for the remaining vector-vector operations is proportional to n. The total costs of each iteration are given by $c_1 s\,e + c_2 n + c_3$ with the constants of time $c_1, c_2, c_3 \in \mathbb{R}$. c_3 considers scalar operations and can be neglected for large sparse matrices. Thus, the contribution of the operations of processor k to the total number of operations is given by $(s\,e_k + \xi n_k)/(s\,e + \xi n)$ with $\xi = c_2/c_1$. The parameter ξ firstly depends on the number of vector operations (without regarding the matrix-vector multiplications). Secondly, it considers the execution times of arithmetical, logical, and memory operations on the processor used. Therefore, it is dependent on both the iterative algorithm and the processor architecture.

For computational load balance, each processor has to perform the p-th fraction of the total number of operations. Hence, the rows of the matrix and the vector components are distributed

according to (1).

$$n_k = \begin{cases} \min\limits_{1 \le t \le n-g_k+1} \left\{ t \,\middle|\, \dfrac{s\,e_k(t) + \xi t}{s\,e + \xi n} \ge \dfrac{1}{p} \right\} & \text{for} \quad k = 0, 1, \ldots, q \\ n - \sum\limits_{i=0}^{q} n_i & \text{for} \quad k = q+1 \\ 0 & \text{for} \quad k = q+2, \ldots, p-1 \end{cases} \tag{1}$$

For large sparse matrices and $p \ll n$, usually $q = p-1$ or $q+1 = p-1$ hold. It should be noted 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 negligible compared with the execution time of s matrix-vector products. In the second case, the execution time of the matrix-vector multiplications hardly contributes to the total execution time.

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

$$a_{\mathrm{MVP}} \approx \frac{s\,e}{s\,e + \xi n} = \frac{1}{1 + \xi/(s\,z_{\mathrm{mean}})}\,. \tag{2}$$

Here, $z_{\mathrm{mean}} = e/n$ is the mean number of non-zeros per row. Additionally, (2) provides a means for measuring ξ. If a_{MVP} is determined by timings, then an approximation to ξ can be computed by

$$\xi \approx \left(\frac{1}{a_{\mathrm{MVP}}} - 1 \right) s\,z_{\mathrm{mean}}\,.$$

On the PARAGON processor i860 XP, the timings result in an approximative value ξ of about 8 for the CG method according to Algorithm 2.1. For CG with polynomial and incomplete Cholesky preconditioning, this value also is applicable since the Chebyshev iteration from Fig. 2.2 requires about the same number of vector-vector operations per matrix-vector product as the basic CG iteration and the operations of the incomplete factorization as well as of the forward/back-substitution are completely local and balanced (see 2.3 and 4.3).

Fig. 2 illustrates the distribution of `col_ind` from Fig. 1 as well as the distribution of the vectors x and y of the matrix-vector multiplication $y = \boldsymbol{A}x$ to four processors for $\xi = 8$ and $s = 1$.

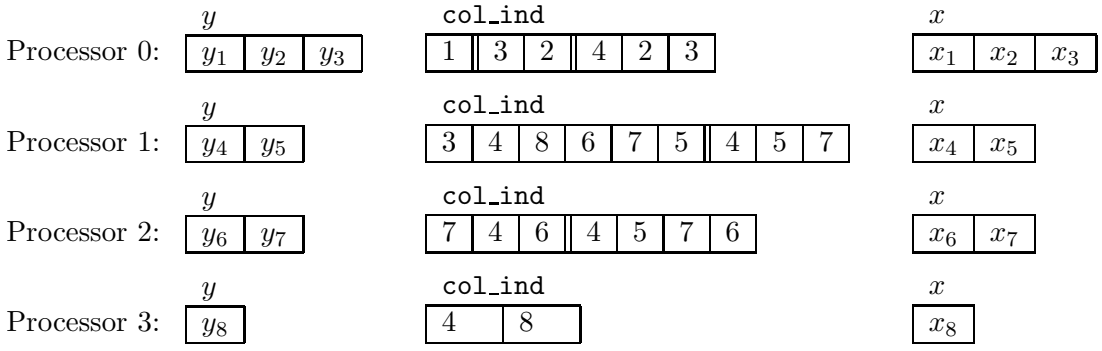

Figure 2: Data distribution for $\xi = 8$ and $s = 1$

4.2 Communication scheme and reordering

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 `col_ind` are analyzed on each processor k to determine which elements result in access to non-local data. Then, the processors exchange information to decide which local data must be sent to which processors. If the matrix-vector multiplication is performed row-wise, components of the vector x of $y = \boldsymbol{A}x$ are communicated. After the analysis, `col_ind` and `value` are reordered in such a way that the data that results in access to processor h is collected in block h. The elements of block h succeed one another row-wise with increasing column index per row. Block k is the first block in the arrays `col_ind` and `value` of processor k. Its elements result in access to local data; therefore, in the following, it is called the local block. The goal of this reordering is to perform computation and communication overlapped. Fig. 3 shows the reordering for the array `col_ind` of processor 1 (see Fig. 2).

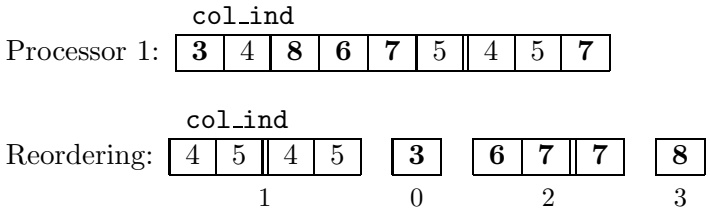

Figure 3: Reordering into blocks

The elements of block 1, the local block, result in access to the local components 4 and 5 of x during the row-wise matrix-vector multiplication, whereas operations with the elements of the blocks 0, 2, and 3 require communication with the processors 0, 2, and 3, respectively. For the parallel matrix-vector multiplication, each processor first executes asynchronous receive-routines to receive necessary non-local data. Then all components of x that are needed on other processors are sent asynchronously. While the required data is on the network, each processor k performs operations with block k. After that, as soon as non-local data from processor h arrives, processor k continues the matrix-vector multiplication by accessing the elements of block h. This is repeated until the matrix-vector multiplication is complete. Computation and communication are performed overlapped so that waiting times are reduced.

The data distribution and the communication scheme we present here do not require any knowledge about a specific discretization mesh; the schemes are determined automatically by the analysis of the indices of the non-zero matrix elements.

4.3 Parallel preconditioners

In this section, we discuss properties of the developed parallel preconditioners with respect to the implementation on massively parallel machines.

4.3.1 Polynomial preconditioning

The advantage of this preconditioner is the reuse of algorithmic elements of the basic CG. Since polynomial preconditioning only requires vector-vector operations and matrix-vector products with $\boldsymbol{A}$, the same parallelization strategies as for the basic CG can be exploited.

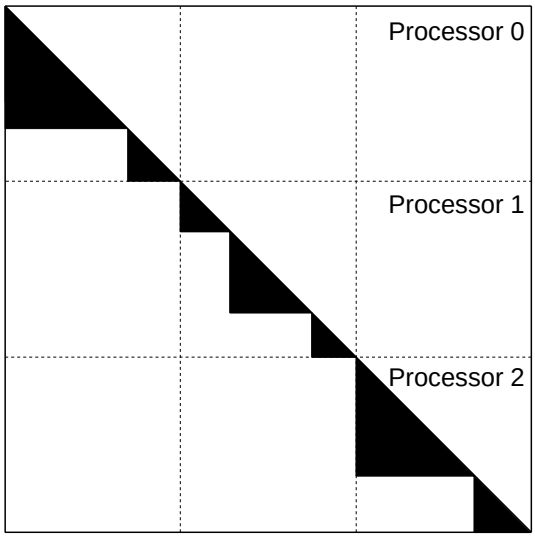

Figure 4: Division into smaller Blocks

4.3.2 Incomplete Cholesky preconditioning

For the incomplete Cholesky preconditioner, we consider quadratic diagonal blocks from the original matrix $\boldsymbol{A}$ (see 2.3).

The data distribution and the reordering strategy from 4.1 and 4.2 result in so-called local blocks at the diagonal positions. If we select the whole local block for the preconditioning, the extra work in the factorization and the forward/back-substitution is often greater than the gain obtained by the reduced number of iterations. Furthermore, the operations for the preconditioning are not balanced because the number of elements in the local blocks varies. Hence we divide the local block into smaller blocks in such a way that the total number of elements in the smaller blocks of each processor is nearly the same (Fig. 4).

The complete work for the preconditioning can be performed without communication if the blocks do not overlap with local blocks of other processors. This would cause additional communication in the factorization part and for calculating the solution of the preconditioning system.

At the beginning of the calculation, the highest number of elements in a small block is determined. This number depends on the machine and the problem's condition. First we determine the block dimension where the forward/back-substitution has the best MFLOPS rate on the computer system used. The number of elements in the lower triangular system with this dimension is the machine dependant parameter η. On the Paragon XP/S 10, a system with dimension 200 achieves the best MFLOPS rate, i.d., $\eta = 20100$.

The percentage of the non-zeros in the skyline of the local block per processor k is the problem dependant parameter ζ_k. If the non-zeros of the local block are closely packed in its skyline, we assume that a lot of information is stored close to the diagonal of the matrix. Otherwise, if the non-zero structure of the local block's skyline is very sparse, we assume that most information of the system is outside the local blocks.

The parameter η is once calculated for the machine used. The parameter ζ_k, however, is computed at runtime by each processor. The product $\eta\zeta_k$ gives the basic value for the number of elements in a lower triangular subblock of processor k's local block. With this basic value each

processor determines the number of elements that are used for the local preconditioner. After this, one global synchronization is necessary for determining the whole number of preconditioning elements e_{prec}, and each processor can calculate its variance from the average value e_{prec}/p. For a good load balance, the basic value is updated so that each processor gets nearly the same number of preconditioning elements.

The preconditioner is stored in row-wise SKS format since during the factorization fill-in is only possible within the skyline. To improve the performance per processor, the elements of the lower triangular preconditioning matrix are additionally stored in reverse ordering for the back-substitution.

5 Results

The numerical and performance tests of the developed parallel preconditioned CG algorithms are performed on the distributed memory system PARAGON XP/S 10 of the Research Centre Jülich. The PARAGON XP/S 10 has 138 processors, each with a 32 Megabyte private memory, interconnected by a two-dimensional mesh. The maximum transfer rate is 200 Megabyte/second per channel in both directions.

5.1 Numerical test cases

The numerical and performance tests we present here are carried out with spd matrices from various application problems. The matrix **FLOW3D** comes from a FE model of environmental science in which the behavior of pollutants in geological systems is simulated [15, 16]. The matrix **STRUCT** stems from a structural mechanics model in which stresses in materials induced by thermal expansion are calculated by applying the FE program SMART [20]. The matrices **PRESS**, **BLOCK**, **TURB**, and **CHEM** originate in simulation models of automobile industry (pressing hoods), structural and mechanical loads, as well as chemistry (conductivity of sintered materials), respectively.

Table 1 shows numerical data of all the considered coefficient matrices. $z_{\max}$ denotes the maximum number of non-zeros per matrix row. The euclidean condition number is estimated by using the approximations of the extreme eigenvalues from the polynomial preconditioner (see 2.2).

Table 1: Numerical data of the considered large sparse matrices

	PRESS	**STRUCT**	**FLOW3D**	**BLOCK**	**TURB**	**CHEM**
Rows	13860	25222	49392	49824	66456	125000
Non-zeros	661010	3856368	1242814	7070824	3380524	860000
$z_{\max}$	49	485	27	243	84	7
Condition	$\approx 10^9$	$\approx 10^5$	$\approx 10^3$	$\approx 10^6$	$\approx 10^6$	$\approx 10^6$

The start vector x_0 for the iterative methods considered is either given by the FE models or computed by

$$(x_0)_j = \frac{(b)_j}{a_{j,j}}, \quad j = 1, \ldots, n$$

where $a_{j,j} \neq 0$, $j = 1, \ldots, n$, denote the diagonal elements of $\boldsymbol{A}$ and $(x_0)_j$ the j-th component of x_0.

For the test cases where no right hand side is given it is chosen as

$$(b)_j = \sum_{i=1}^{n} a_{j,i}, \quad j = 1, \ldots, n$$

so that the exact solution vector is $x^* = (1, \ldots, 1)^T$.

As stopping criterion for the iterative algorithms we use the maximum scaled absolute difference of the components of the solution vector's latest two approximations:

$$\max_{j=1,\ldots,n} 2\frac{|(x_{i+1})_j - (x_i)_j|}{|(x_{i+1})_j| + |(x_i)_j|} \leq \varepsilon_s \,. \tag{3}$$

For $|(x_{i+1})_j|$ and $|(x_i)_j|$ less than ε_s, the denominator in (3) is set to ε_s. In all test cases, the iteration is stopped when the maximum scaled absolute difference is less than 10^{-10}.

5.2 Performance results

On the PARAGON, the tests are performed using the PARAGON FORTRAN compiler, version 4.5 [18, 19], and the PARAGON OSF/1 operating system, release 1.2 [17] that enables the use of the communication processor. The programs are compiled with the optimization switches *-O4 -Knoieee*.

Table 2 shows execution times in seconds and the corresponding numbers of iterations displayed in parantheses for all the test cases on up to 128 PARAGON processors. Only the test case **PRESS** is small enough with respect to memory requirements to be executed on one processor. **FLOW3D**, **STRUCT**, **CHEM**, **BLOCK**, and **TURB** require the memory of more than one processor. All the following speedup considerations refer to the execution time on the least processor number on which the measurement is performed. Up to this processor number, we assume optimum linear speedup, e.g. a speedup of 8.0 for **STRUCT** on eight processors. Since the incomplete Cholesky preconditioned method requires additional memory for the matrix factorization, the first timings are performed on 16 and eight processors for **BLOCK** and **TURB**. For all measurements with polynomial preconditioning, a degree $m = 2$ of the polynomial is chosen except for **PRESS**; in the latter case m is set to 8. Polynomial preconditioning is always combined with diagonal scaling, i.e., it is applied to the diagonally scaled coefficient matrix.

For CG with diagonal scaling, the iteration numbers in Table 2 slightly vary for different processor numbers. This is mainly caused by different rounding errors due to the segment-wise computation of vector reductions. The less the number of iterations the smaller usually is the influence of numerical errors. For the polynomially preconditioned method, the iteration numbers are reduced by nearly a factor of $m + 1$ compared with the diagonally scaled CG and are much more stable for varying processor numbers. The higher the degree of the polynomial the more robust is the method with respect to numerical errors. For CG with incomplete Cholesky preconditioning, the iteration numbers vary — depending on the non-zero pattern of the matrix — more markedly for different processor numbers since the diagonal blocks used for the preconditioner differ with increasing processor numbers. However, incomplete Cholesky preconditioning considerably decreases the iteration numbers and hence usually results in a very robust iteration.

For the best conditioned matrix **FLOW3D**, diagonally scaled CG gives the lowest execution times since computing the preconditioner is very cheap (see also [12]). For the other markedly worse conditioned matrices, one of the more sophisticated preconditioning techniques results in shorter times than those of diagonally scaled CG. Furthermore, all three methods considered show an advantageous scaling behavior that improves with increasing problem size. For the largest test case

Table 2: Timings in seconds and iterations displayed in parentheses

FLOW3D	p=1	p=2	p=4	p=8	p=16	p=32	p=64	p=128
Diag. scal.	-	47.2	24.8	13.0	7.1	4.0	2.3	1.4
	-	(144)	(144)	(144)	(144)	(144)	(144)	(144)
Pol. prec.	-	51.2	27.2	14.3	7.9	4.5	2.6	1.5
	-	(61)	(61)	(61)	(61)	(61)	(61)	(61)
Inc. Chol.	-	260	85.9	30.3	13.4	7.0	2.9	1.7
	-	(125)	(133)	(129)	(130)	(106)	(87)	(99)
STRUCT	p=1	p=2	p=4	p=8	p=16	p=32	p=64	p=128
Diag. scal.	-	-	-	146	79.3	42.1	22.6	13.5
	-	-	-	(887)	(903)	(914)	(905)	(905)
Pol. prec.	-	-	-	171	90.7	47.3	25.4	14.9
	-	-	-	(357)	(357)	(357)	(357)	(357)
Inc. Chol.	-	-	-	120	60.3	31.2	16.9	9.7
	-	-	-	(555)	(513)	(473)	(467)	(487)
PRESS	p=1	p=2	p=4	p=8	p=16	p=32	p=64	p=128
Diag. scal.	12002	6070	3161	1634	863	483	314	235
	(47040)	(46568)	(46660)	(46763)	(46433)	(46729)	(46472)	(46963)
Pol. prec.	11343	5929	3071	1570	818	436	266	172
	(5954)	(5954)	(5941)	(5933)	(5933)	(5881)	(5881)	(5954)
Inc. Chol.	709	335	209	153	82.7	41.3	31.1	29.0
	(1575)	(1532)	(1870)	(2698)	(2807)	(2600)	(3420)	(4513)
CHEM	p=1	p=2	p=4	p=8	p=16	p=32	p=64	p=128
Diag. scal.	-	1148	606	301	166	92.9	56.7	31.6
	-	(2324)	(2324)	(2229)	(2320)	(2324)	(2320)	(2229)
Pol. prec.	-	862	465	244	131	75.4	46.9	27.1
	-	(789)	(789)	(797)	(797)	(797)	(797)	(797)
Inc. Chol.	-	2966	1229	547	256	134	81.8	55.8
	-	(1954)	(2031)	(2040)	(1971)	(2027)	(1902)	(1975)
BLOCK	p=1	p=2	p=4	p=8	p=16	p=32	p=64	p=128
Diag. scal.	-	-	-	470	237	122	64.8	36.6
	-	-	-	(1561)	(1561)	(1561)	(1560)	(1561)
Pol. prec.	-	-	-	472	238	122	65.4	37.0
	-	-	-	(548)	(548)	(548)	(548)	(548)
Inc. Chol.	-	-	-	-	239	121	63.8	34.7
	-	-	-	-	(1175)	(1160)	(1136)	(1111)
TURB	p=1	p=2	p=4	p=8	p=16	p=32	p=64	p=128
Diag. scal.	-	-	3859	2032	1138	561	319	176
	-	-	(11039)	(11041)	(11042)	(11042)	(11038)	(11038)
Pol. prec.	-	-	3630	1907	1085	534	305	172
	-	-	(3868)	(3868)	(3868)	(3868)	(3868)	(3868)
Inc. Chol.	-	-	-	1636	882	411	230	126
	-	-	-	(5820)	(5694)	(5387)	(5387)	(5293)

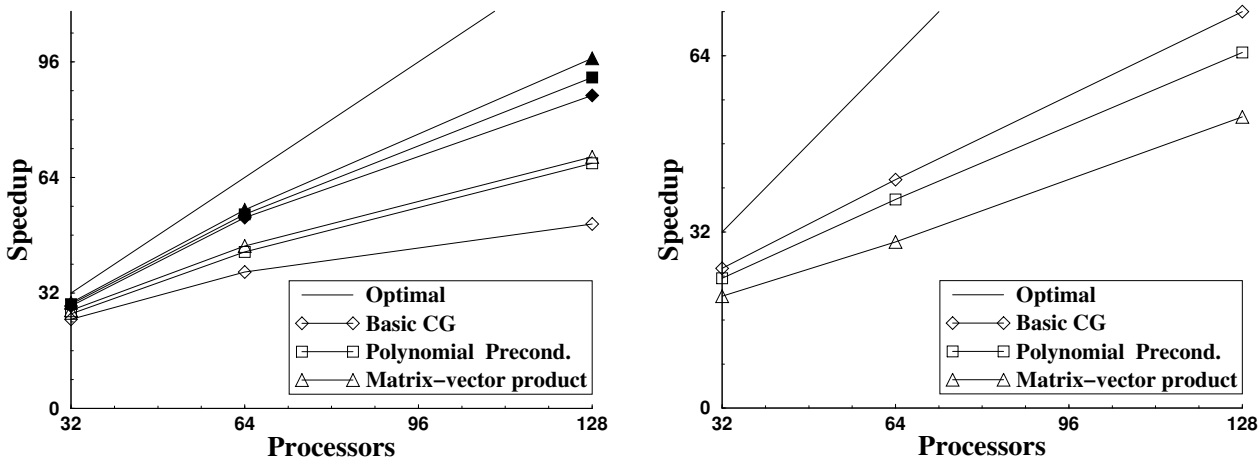

Figure 5: Speedup of matrix-vector multiplication and CG with diagonal scaling as well as polynomial preconditioning. Left: **STRUCT** (filled markers) and **PRESS** (non-filled); right: **CHEM**.

BLOCK, we achieve speedups of 102.7, 102.1, and 110.2 on 128 processors for CG with diagonal scaling, polynomial preconditioning, and incomplete Cholesky preconditioning, respectively.

In the following, we describe the particular properties of the developed parallel CG methods with polynomial and incomplete Cholesky preconditioning.

5.2.1 Polynomial preconditioning

The basic CG as well as the developed preconditioned methods require one synchronization point per iteration due to vector reductions. The necessary global communication markedly decreases the CG speedup on large numbers of processors.

The gap between the CG speedup and the matrix-vector product speedup in Fig. 5 on the left is caused by this synchronization overhead. The size of the gap depends on the matrix pattern and the number of processors used. Polynomial preconditioning requires additional matrix-vector products, but decreases vector reductions such as inner products. Therefore the speedup of CG with polynomial preconditioning approaches to the matrix-vector product speedup the closer, the higher the degree of the polynomial is chosen. The optimum degree depends on the gap size and increases with the number of processors used [14]. If the gap between the matrix-vector product and CG speedup is small, e.g. for **STRUCT**, only a weak scalability improvement is possible, and small degrees are suited. In the case of larger gaps, as for **PRESS**, higher degrees become attractive since they improve the scaling behavior and result in shorter execution times (see Table 2). This observation was confirmed for various other examples [14].

Fig. 5 on the right shows a different behavior for the problem **CHEM**. The small number of non-zeros per row results in a matrix-vector product with a bad calculation/communication ratio. However, the vector-vector operations scale very well due to the large order of the matrix. Thus, the CG speedup is better than the matrix-vector product speedup. With polynomial preconditioning, the speedup again approaches to the matrix-vector product speedup, but this time it is worse than that of the basic CG. Hence, a small degree is suited to keep a good scalability. Note that for **CHEM** CG with polynomial preconditioning has the best time behavior of all three methods considered due to the reduced number of vector reductions and the low costs for the matrix-vector computations.

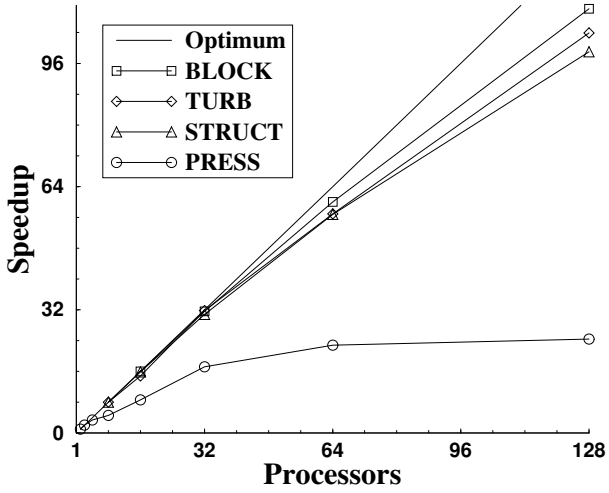

Figure 6: Speedup of CG with incomplete Cholesky preconditioning

5.2.2 Incomplete Cholesky preconditioning

The scalability of CG with parallel incomplete Cholesky preconditioning is displayed in Fig. 6 for the matrices **BLOCK**, **TURB**, **STRUCT**, and **PRESS** on up to 128 processors.

For the large problems **STRUCT**, **TURB**, and **BLOCK**, high speedups of 99.0, 103.9, and 110.2 are achieved on 128 processors. For these matrices, the preconditioning effect is nearly the same for all processor numbers used in the investigations. The number of iterations in Table 2 even decreases slightly with increasing processor number since the non-zero structure close to the diagonal is compact. Hence the local blocks become smaller but also denser with increasing processor number, and thus more elements are considered for the preconditioner (see 4.3). This improves the preconditioning effect.

For the relatively small matrix **PRESS**, also with a compact non-zero structure around the diagonal, the iterations vary from 1575 on 1 processor to 4513 on 128 processors. This causes the worse speedup behavior in Fig. 6. For **PRESS**, the preconditioning effect decreases markedly since the local blocks become too small for high processor numbers, and thus considerably less elements are taken for the preconditioner. Nevertheless, the execution time on one processor is 16 times faster than that for CG with polynomial preconditioning and still 5.9 times faster on 128 processors.

For the matrix **CHEM**, the developed incomplete Cholesky preconditioner does not work well since most non-zeros lie outside the diagonal area and are not considered for the preconditioner.

For **FLOW3D**, also many significant elements are not inside the very sparse local blocks. Different from the other test cases, the contribution of the Cholesky factorization to the total execution time is high for small processor numbers. On two processors, e.g., the decomposition takes 75% of the total execution time. For higher processor numbers, the number of preconditioning elements is markedly decreased since the local blocks become smaller. Thus the operational costs for the factorization considerably sink with increasing processor number. This causes the superlinear speedup for **FLOW3D** in Table 2.

6 Conclusions

We presented parallel preconditioned CG methods and demonstrated by case studies with matrices of different sparsity from real applications that the developed data distribution and the communication scheme together with the reduction of synchronization do result in an advantageous scaling behavior of the algorithms on massively parallel machines.

For well-conditioned matrices, it is hard to beat diagonal scaling by more sophisticated preconditioners since this method is effective and cheap to compute. Polynomial preconditioning shows the best results if the coefficient matrix is so sparse that matrix-vector multiplications do not dominate the solver's total execution time. Especially for high processor numbers, this preconditioner markedly reduces synchronization costs. Moreover, the preconditioning effect is stable for varying processor numbers. Opposite to polynomial preconditioning, the developed fully parallel incomplete Cholesky preconditioner performs best if matrix-vector operations dominate the iteration's total execution time. The preconditioning effect is the better, the more significant matrix elements are close to the diagonal. These two conditions are usually given for matrices from structural mechanics applications so that CG with parallel incomplete Cholesky preconditioning shows the most advantageous time behavior for this kind of matrices in our investigations.

References

[1] St. Ashby, Minimax polynomial preconditioning for Hermitian linear systems, *SIAM J. Matrix Anal. Appl.* 12 (1991) 766–789.

[2] O. Axelsson, *Iterative Solution Methods* (Cambridge University Press, 1994).

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

[4] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* (SIAM, Philadelphia, 1993).

[5] A. Basermann, Conjugate gradients parallelized on the hypercube, *International Journal of Modern Physics C* 4 (1993) 1295–1306.

[6] A. Basermann, Parallel sparse matrix computations in iterative solvers on distributed memory machines, in: D.H. Bailey et al., eds., *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing* (SIAM, Philadelphia, 1995) 454–459.

[7] A.T. Chronopoulos and C.W. Gear, s-step iterative methods for symmetric linear systems, *J. Comput. Appl. Math.* 25 (1989) 153–168.

[8] P. Deuflhard and A. Hohmann, *Numerische Mathematik* (de Gruyter, 1991).

[9] G.H. Golub and C.F. Van Loan, *Matrix Computations* (The Johns Hopkins University Press, Baltimore, second edition, 1989).

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

[11] J.M. Ortega, *Introduction to parallel and vector solution of linear systems* (Plenum Press New York, London, 1988).

[12] G. Pini and G. Gambolati, Is a simple diagonal scaling the best preconditioner for conjugate gradients on supercomputers?, *Adv. Water Resources* 13 (1990) 147–153.

[13] T.J. Rivlin, *The Chebyshev Polynomials* (John Wiley, 1974).

[14] C. Schelthoff and A. Basermann, Polynomial preconditioning for the conjugate gradient method on massively parallel systems, in: K. Ecker, J. Apsel, eds., *Workshop über Parallelverarbeitung* (Informatik-Bericht 95/1, TU Clausthal, Institut für Informatik, 1995) 150–167.

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

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

[17] Paragon User's Guide, Intel Supercomputer Systems Division, Beaverton, Oregon, June 1994.

[18] Paragon Fortran Compiler User's Guide, Intel Supercomputer Systems Division, Beaverton, Oregon, March 1994.

[19] Paragon Fortran Compiler, Release 4.5, Software Product Release Notes, Intel Supercomputer Systems Division, Beaverton, Oregon, March 1994.

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