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

Interner Bericht

Polynomial Preconditioning for the Conjugate Gradient Method on Massively Parallel Systems

Christof Schelthoff, Achim Basermann

KFA-ZAM-IB-9423

Oktober 1994
(Stand 31.10.94)

Tagungsband des Workshops über Parallelverarbeitung, 25.September-1.Oktober 1994, Lessach, Österreich

Polynomial Preconditioning for the Conjugate Gradient Method on Massively Parallel Systems

Christof Schelthoff [1]
Achim Basermann [2]
Central Institute for Applied Mathematics
Research Centre Jülich
52425 Jülich, Germany

Abstract

A frequently used iterative algorithm for solving large, sparse, symmetric and positive definite systems of linear equations is the method of conjugate gradients (CG). This method requires one matrix-vector product and some dot products in each iteration. Convergence is dependent on the condition number of the coefficient matrix. So preconditioning techniques are used to reduce the number of iterations.

In this context, polynomial preconditioning was developped. This method decreases the total number of dot products by reducing the total number of iterations. Of course, some additional work has to be done for the preconditioning. When a polynomial of degree k is used, k matrix-vector products per iteration have to be calculated rather than one. On scalar machines, this shift between matrix-vector products and dot products influences the performance of the algorithm only slightly. On massively parallel systems, dot products require global synchronization, while the calculation of matrix-vector products merely results in communication with a small number of processors. The actual implementation used here is based on Chebyshev polynomials. Performance tests were carried out on the Intel Paragon XP/S 10 with 140 nodes at the Research Centre Jülich (KFA). The CG method with polynomial preconditioning shows better performance and scalability than the basic method on massively parallel machines. Additionally there are some numerical advantages like a higher accuracy and an increased stability.

Keywords: Sparse matrices; Conjugate gradients; Polynomial preconditioning; Chebyshev polynomials; Parallelization; Distributed memory

1 Introduction

Many physical problems result in the solution of partial differential equations. An exact solution can not be found in general. So we consider a discrete solution. Using difference methods or finite element methods we obtain large sparse systems of linear equations. Often the systems are symmetric and positive definite (spd). A frequently used method to solve these systems is the conjugate gradient method (CG). Since convergence depends on the eigenvalue distribution of the coefficient matrix, we use preconditioning techniques to accelerate convergence.

Parallelizing the CG method leads to the question which preconditioners are suitable to massively parallel machines. A well preconditioned CG

[1]E-Mail: ch.schelthoff@kfa-juelich.de
[2]E-Mail: a.basermann@kfa-juelich.de

must at least scale as well as the basic CG. We present a method originally developped by Rutishauser [15] to improve the stability of CG towards rounding errrors. The advantage of polynomial preconditioning is that the total number of iterations is dynamically reduced by varying the degree of the polynomial. High degrees usually results in a small number of iterations. This reduces rounding errors in CG that mainly result from the orthogonalizations that have to be performed once in each iteration step.

Another reason for using polynomial preconditioning is the pleasant fact that it is easy to expand an existing CG algorithm with this preconditioner independent of the specific implementation.

2 The Conjugate Gradient Method

To solve the sparse linear system

$$Ax = f,$$

we can use the conjugate gradient method if the $n \times n$-matrix A is symmetric ($A^T = A$) and positive definite ($x^T Ax > 0 \quad \forall\ x \in \mathrm{I\!R}^n \backslash \{\vec{0}\}$). The latter condition is equivalent to the statement that all eigenvalues λ_i of A are positive.

In the following, we consider the residual $r(x) = f - Ax$, because for regular A the following equivalence holds:

$$r(x) = \vec{0} \qquad \Longleftrightarrow \qquad Ax = f.$$

So the main intention is to find the point, where the residual is the zero-vector. Using the residual, the real function

$$\begin{aligned} F(x) \ &:= \ \frac{1}{2} r(x)^T A^{-1} r(x) \\ &= \ \frac{1}{2} x^T Ax - f^T x + \frac{1}{2} f^T A^{-1} f \ , \end{aligned}$$

can be defined with the properties

- $F(x) > 0 \quad \forall\ r(x) \neq \vec{0}$,
- $F(x) = 0 \quad r(x) = \vec{0}$.

Note that if A is spd so is A^{-1}. Thus, the minimum x^* of $F(x)$ is the solution of the system $Ax = f$. Secondly, we can calculate the gradient of this function

$$\mathrm{grad}\{F(x)\} \ = \ Ax - f = -\ r(x)$$

that is the negative residual. Since A is regular the gradient is only zero for the global minimum $x = x^*$, and there are no other local minima. The classical way to find the minimum of a function is the iteration

$$x_{j+1} = x_j + \alpha_j p_j \qquad j = 0, 1, \ldots$$

p_j is called the search direction, α_j the step width.

Once the search direction has been determined, we obtain the optimum α_j for $F(x)$ applying line search

$$\alpha_j = \frac{r_j^T p_j}{p_j^T A p_j} \ ,$$

with $r_j := r(x_j)$.

Using the negative gradient as search direction p_{j+1} that is A–orthogonalized to all previous search directions $p_0, \ldots, p_j$ [7][16] we obtain the *conjugate gradient method for spd linear systems.* In that case, $r_j^T p_j = r_j^T r_j$ holds and therefore

$$\alpha_j = \frac{r_j^T r_j}{p_j^T A p_j} \ .$$

In each iteration step, the orthogonalization is calculated by

$$p_{j+1} = r_{j+1} + \beta_j p_j$$

with $\beta_j = \frac{r_{j+1}^T r_{j+1}}{r_j^T r_j}$. Aykanat et al. [2] suggested an alternative calculation of β_j using

$$\beta_j = \alpha_j \frac{(Ap_j)^T A p_j}{p_j^T A p_j} - 1$$

that allows to calculate β_j without knowing the new residual r_{j+1}. Thus, we can calculate α_j and β_j successively. The modified CG algorithm is displayed in fig. 1.

On parallel machines, reductions — the dot products — result in global synchronization. In the parallel case, the modified algorithm requires only one synchronization point rather than two in the basic method [2].

In each iteration of the CG method, the following operations have to be computed:

- 1 matrix-vector product,
- 3 successive dot products,
- 3 vector additions,
- some scalar operations.

```
Init.: $x_0,\ r_0 := f - Ax_0,\ p_0 := r_0$
For $j := 0, 1, \ldots$
    $\alpha_j = \frac{r_j^T r_j}{p_j^T A p_j}$
    $\beta_j = \alpha_j \frac{(Ap_j)^T A p_j}{p_j^T A p_j} - 1$
    $x_{j+1} = x_j + \alpha_j p_j$
    $r_{j+1} = r_j - \alpha_j A p_j$
    $p_{j+1} = r_{j+1} + \beta_j p_j$
until convergence
```

Figure 1: Modified CG algorithm

2.1 Convergence Properties

The convergence of the CG method depends on the eigenvalue distribution of A [12]. A criterion for the width of the spectrum is the euclidean condition number that is for spd matrices

$$\kappa_2(A) := \frac{\lambda_{max}}{\lambda_{min}} \quad (\geq 1)\ .$$

With $\gamma := \frac{\sqrt{\kappa_2(A)}-1}{\sqrt{\kappa_2(A)}+1}$ the error in the $j-$th iteration is bounded by

$$\| x_j - x^* \| \leq 2\sqrt{\kappa_2(A)}\gamma^j \| x_0 - x^* \| \ .$$

The right hand side increases if the condition number increases. So lower condition numbers usually accelerate convergence.

3 Preconditioning

The convergence result leads to the idea that a transformation that decreases the condition number accelerates the convergence. In other words, we want to find a matrix C — the preconditioner — to perform the following transformation

$$CAx \;=\; Cf.$$

The aim is to obtain a new implicit coeffic*ent matrix CA with small $\kappa(CA)$. Implicit means that we do not evaluate the matrix product $C{\cdot}A$ explicitly, but we replace the matrix-vector product Ap_j by $C \cdot (Ap_j)$ in the CG-algorithm of fig. 1. That means calculating additional matrix-vector products. Further, f is replaced by $C \cdot f$ in the initialization.

To answer the question which C are suitable, we have to deal with the following problems. First, we have the storage problem for C. Secondly, we have to guarantee the assumptions of CG, that is, CA must be again symmetric and positive definite. In this context, polynomial preconditioning is well suited. Here C is a polynomial of degree m in A, that means

$$C = \mathcal{C}(A) \;=\; \sum_{i=0}^{m} \nu_i A^i \ .$$

We obtain the new system

$$\underbrace{\mathcal{C}(A)A}_{\mathcal{P}(A)}\,x \;=\; \mathcal{C}(A)f.$$

So $\mathcal{P}(A)$ is a polynomial of degree $k := m+1$ with $\mathcal{P}(0) = 0$, and we achieve the CG algorithm with polynomial preconditioning in fig. 2.

Of course the new coeffic*ent matrix $\mathcal{P}(A)$ is symmetric. In each iteration we now have to calculate $\mathcal{P}(A)p_j$, so there are k matrix-vector products rather than one in the basic method. These matrix-vector products only use the original matrix A and thus no additional storage is required. Finally, we have to guarantee that $\mathcal{P}(A)$ only has positive eigenvalues.

It is easy to see that if λ_i is an eigenvalue of A, $\mathcal{P}(\lambda_i)$ is the corresponding eigenvalue of $\mathcal{P}(A)$. However, we do not know the eigenvalues in general. Therefore we use the stronger criterion $\mathcal{P}(\xi) > 0 \quad \forall\ \xi \in [\lambda_{min}, \lambda_{max}]$ or $\forall\ \xi \in [0, \infty)$ to guarantee that $\mathcal{P}(A)$ is positive definite.

By polynomial preconditioning of degree k, the number of iterations of the basic CG can be reduced as follows [3]:

If the basic CG needs i_{CG} steps, the CG with polynomial preconditioning requires at least i_{CG}/k steps.

Since we have to calculate k matrix-vector products in each step of the preconditioned CG rather than one in each step of the basic CG, the total number of matrix-vector products increases. However, we only have to

```
Init.: $x_0,\ r_0 := \mathcal{C}(A)\cdot(f - Ax_0),\ p_0 := r_0$
For $j := 0, 1, \ldots$
    $\alpha_j = \frac{r_j^T r_j}{p_j^T \mathcal{P}(A) p_j}$
    $\beta_j = \alpha_j \frac{(\mathcal{P}(A)p_j)^T \mathcal{P}(A) p_j}{p_j^T \mathcal{P}(A) p_j} - 1$
    $x_{j+1} = x_j + \alpha_j p_j$
    $r_{j+1} = r_j - \alpha_j \mathcal{P}(A) p_j$
    $p_{j+1} = r_{j+1} + \beta_j p_j$
until convergence
```

Figure 2: CG algorithm with polynomial preconditioning

perform one synchronization in each step of both the basic and the preconditioned method. So the reduction of iteration steps leads to less global synchronization. Increasing the degree of the polynomial increases the number of matrix-vector products but decreases the total number of global synchronizations.

Finally, we have to answer which polynomials are suitable. Remember, our aim was to obtain a small $\kappa_2(\mathcal{P}(A))$ that is close to $\kappa_2(I) = 1$. So we formulate the following minimization problem for P_k — the polynomials of degree k — with $\mathcal{P}(0) = 0$:

$$\min_{\mathcal{P}\in P_k} \| \, I - \mathcal{P}(A) \, \| \ .$$

The solution is norm-dependent. Here we use the spectral norm and obtain because $I - \mathcal{P}(A)$ is symmetric

$$\min_{\mathcal{P}\in P_k} \quad \max_{\lambda_i \text{ EV of } A} | \, 1 - \mathcal{P}(\lambda_i) \, | \ .$$

Here we want to find such a polynomial that transforms the original eigenvalues of A so that they are as close as possible to 1.

Note that in general the eigenvalues are unknown. Thus, we do not consider the discrete problem but the continous version for $0 < a < b$

$$\min_{\mathcal{P}\in P_k} \max_{\lambda\in[a,b]} | \, 1 - \mathcal{P}(\lambda) \, | \ ,$$

where a and b are estimates for λ_{min} and λ_{max}, the extreme eigenvalues of A. The solution of this problem is well known and can be expressed using scaled and translated Chebyshev polynomials [1] [16]

$$\mathcal{P}(\lambda) = 1 - \mathcal{T}_k(\frac{a+b-2\lambda}{b-a})/\mathcal{T}_k(\frac{a+b}{b-a}) \ ,$$

where $\mathcal{T}_k$ is the k-th Chebyshev polynomial [14].

The figures 3 and 4 show some of these polynomials. You see that for even k the function can produce negative eigenvalues if the estimate for the largest eigenvalue is too small. Therefore, we only use odd k that guarantee that $\mathcal{P}(A)$ is again positive definite for arbitrary a and b.

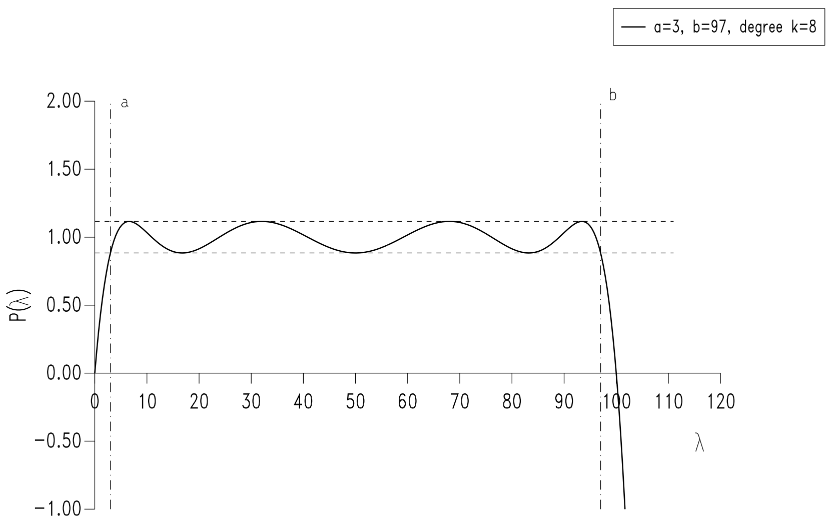

Figure 3: Chebyshev polynomial for even k

At last, we have to describe how the estimates a and b for the extreme eigenvalues can be obtained. The first idea is to use the Gershgorin estimate for the upper bound and a small value for the lower bound (the Gershgorin estimate is not suited, because mostly this value is negative). The problem is that the scaled and translated Chebyshev polynomials are very sensitive to the lower bound. If this bound is close to zero the polynomial oscillates extremely as you see in fig. 5.

Hence we need more exact estimates. The connection of the CG method with the Lanczos method for the real symmetric eigenvalue problem allows to calculate suitable intervals $[a, b]$ within the spectrum of the implicit coefficient matrix $\mathcal{P}(A)$. Of course, we need estimates for A itself. If we use odd k and

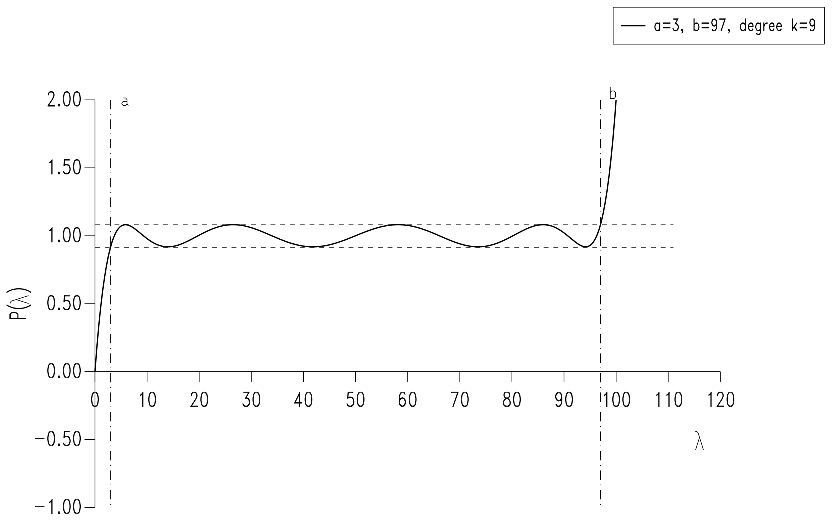

Figure 4: Chebyshev polynomial for odd k

exploit the Lanczos connection for calculating eigenvalues of $\mathcal{P}(A)$ outside the oscillation zone we can determine approximations of the original eigenvalues as you see in fig. 4.

Thus the algorithm becomes adaptive. We compute the extreme eigenvalues of $\mathcal{P}(A)$ after a certain number of steps and check if they are outside of the oscillation zone. If so the corresponding original eigenvalue are calculated. The interval is expanded by updating the values of a and/or b.

A detailed description of the Lanczos connection can be found in [10] and [16]; more details about polynomial preconditioning with Chebyshev and other polynomials are described in [1].

Finally, we should remark that the calculation of the coefficients of the scaled and translated Chebyshev polynomials is not necessary. The *Chebyshev iteration* [8][16], a 3–term recursion, can be applied.

4 Implementation

Polynomial preconditioning is easy to implement when CG is already parallelized. An advantage of this preconditioner is that algorithmic elements of the basic CG can be reused. Since polynomial preconditioning only requires vector additions and matrix-vector products with A, the same parallelization strategies can be exploited as applied for the basic CG.

For our implementation, we use A. Basermanns basic CG [4][5] [6] that

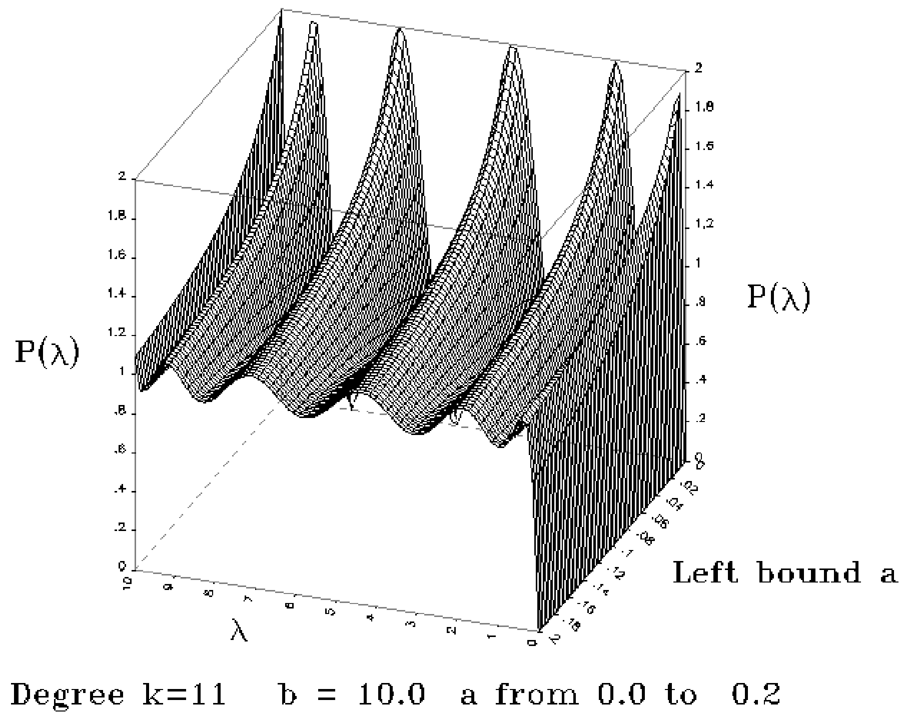

Figure 5: Chebyshev polynomials for different lower bounds a

was implemented on an Intel PARAGON X/PS 10 with 140 nodes at the Research Centre Jülich.

In the following, we describe the implementation of the basic CG because the performance of the preconditioned method mostly depends on the performance of the matrix-vector product already used in the basic CG.

The description is brief because it is only an example how to implement the preconditioner. If you are more interested in parallelizing CG on a distributed memory machine you find a detailed description about data distribution, communication scheme and their application in iterative methods like CG and a Lanczos method in [4][5][6].

Parallelizing CG can be divided into several parts. First we have to distribute the matrix to the processors. Each processor gets a certain number of successive, complete rows and the corresponding components of all vectors. Thus, vector additions can be performed locally without communication. Dot products and reductions, however, require global synchronization.

To balance the computational load in the processors, the rows of the matrix are distributed in such a way that the execution time for each partial iteration is nearly the same on all processors. This is achieved by a suitable data distribution. If we distribute the data to the processors so that each processor gets the same number of rows the vector–vector operations are balanced, but if the matrix is unstructured the matrix-vector product is not balanced. On the other hand, if we distribute the matrix so that each processor gets the same number of nonzeros of the coefficient matrix we have balanced the matrix-vector multiplication but not the vector–vector operations. Hence, we use a distribution that is between these possibilities. Each processor gets so much rows that the total number of arithmetic operations of one partial iteration is (nearly) the same on all processors. This distribution depends on several conditions, in particular on the specific implementation and the machine used . A detailed description can be found in [4] or [6].

For computing a matrix-vector multiplication, you need components of the vector of this operation from other processors. The indices of these components only depend on the sparsity pattern of A. So the communication scheme can be determined before the CG iteration. The matrix-vector product is calculated in the following way. First, each processor asynchronously sends the vector components that are necessary for the partial matrix-vector multiplication on other processors. Then each processor calculates the local part of the matrix-vector product. After that, each processor waits until non–local data arrives and continues the computation of the matrix-vector product using this data. This is repeated until the computation of the matrix-vector product is complete, and the iteration is continued without a global synchronization for the matrix-vector multiplication. Overlapping communication and calculation results in higher performance for the matrix-vector product.

5 Results

In this section, we always consider linear systems that are scaled with the diagonal of the matrix. This diagonal scaling is a simple, frequently used preconditioner.In the following, no preconditioning means only using diagonal scaling.

For the tests, we used the stopping criterion

$$\delta_{scal} = \max_{1 \le i \le n} 2 \frac{|(x_{j+1})_i - (x_j)_i|}{|(x_{j+1})_i| + |(x_j)_i|} < 10^{-5}$$

that is called the *maximum scaled absolute difference* of the components of the latest two approximations of the solution vector. $(x)_i$ denotes the $i-$th component of the vector x.

5.1 Test matrices

The matrices shown in tab. 1 were used for performance tests.

Example matrices	n (rows)	nonzeros	$\kappa_2(A)$
Structural mechanics 1	1,806	63,454	$\sim 10^6$
Structural mechanics 2	10,974	428,650	$\sim 10^7$
Structural mechanics 3	25,222	3,856,386	$\sim 10^5$
Crash Test	13,860	661,010	$\sim 10^9$

Table 1: Examples

The first and the second example stem from the *Harwell Boeing Set of Sparse Matrices*, example *BCSSTK14* and *BCSSTK17* [9]. The other ones were obtained from physical simulation applications at the Research Centre Jülich.

5.2 Numerical Results

We do not want to describe the numerical behaviour of CG in detail. As mentioned above the preconditioned algorithm increases the stability of the method. We observed that iteration numbers of the basic CG for the example *crash test* vary on parallel machines markedly when we changed the number of processors used. Using preconditioning we obtained nearly the same number of steps for different numbers of processors.

A second aspect is that in CG rounding errors are mainly caused by the orthogonalization that requires the computation of dot products. Reducing the iteration numbers by preconditioning we reduce rounding errors, and we get a higher accuracy of the approximation [16].

The question remains if the Chebyshev polynomials are a good choice, since the optimum reduction (by a factor of k) could be far away from realistic results. In fig. 6 we see that the polynomials are well suited for the example *structural mechanics 2*. The optimum and the values measured are very close to each other. The other examples show similar results.

5.3 Performance Results

The speedup of CG is usually determined by the matrix-vector product speedup. However, the CG speedup is lower than the matrix-vector product speedup because of the badly scalable dot products.

The execution time for a dot product is shown for the largest example in fig. 7. We see that for large numbers of processors the communication

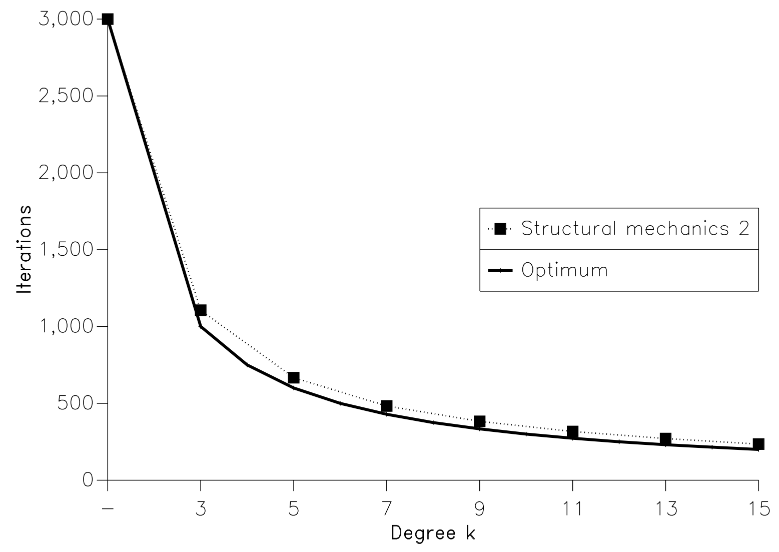

Figure 6: Iterations for different degrees for *structural mechanics 2*

time dominates the time for the dot product and so the time increases if we increase the number of processors. When many processors are used the overhead of the global synchronization increases; the speedup of the operation decreases markedly. If polynomial preconditioning is used less dot products and more matrix-vector products are performed. Thus the CG speedup is closer to the matrix-vector product speedup. Hence increasing the degree of the polynomial increases the speedup. The limiting speedup is the matrix-vector product speedup. However, the speedup — the ratio of the execution times on one and p processors for a certain degree — is not a criterion for the best degree on a fixed number of processors. The criterion is a minimal execution time for a certain degree. With increasing number of processors higher degrees may result in shorter execution times than smaller degrees of the polynomial.

In the following, we demonstrate the principal behaviour of polynomial preconditioning using the (small) example *structural mechanics 1*. The sequential times in seconds for various degrees are given in tab. 2. We see for this matrix that preconditioning does not improve the performance in the sequential case, but time increases only slightly.

The scalability properties for different degrees are displayed in tab. 3. For various numbers of processors, the speedups are shown dependent on the

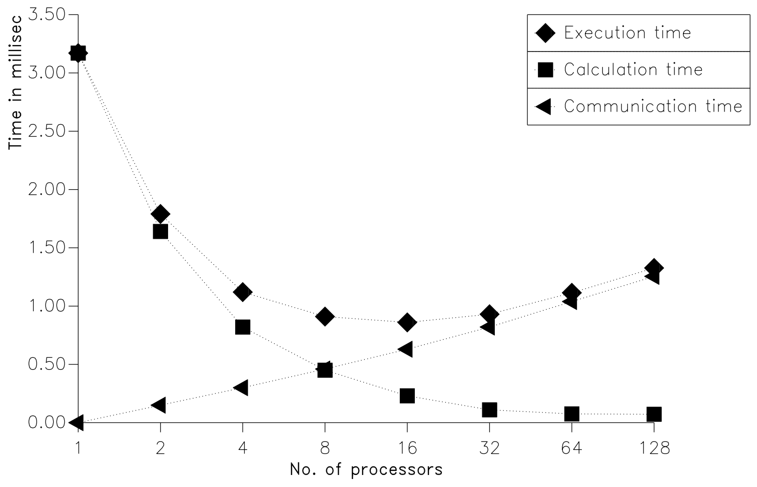

Figure 7: Speedup for one dot product of *structural mechanics 3*

No preconditioning	Degree k=3	Degree k=5
14.6	15.7	16.6

Table 2: Sequential execution times in seconds

degree of the polynomial. In the bottom line, the (limiting) speedup of the matrix-vector product is given.

We see that the gap between matrix-vector product speedup and the speedup of the CG method with no preconditioning increases if the number of processors increases. By using polynomial preconditioning this gap becomes markedly smaller. Tab. 4 shows the corresponding execution times. On 16 and 32 processors, preconditioning already results in shorter execution times.

The behaviour of larger examples is shown in fig. 8, 9 and 10 for a fixed degree. With polynomial preconditioning the gap between the speedup of the matrix-vector product and the CG speedup becomes markedly smaller. Since only speedups are plotted, the best execution times depend on the sequential times.

For *structural mechanics 2*, this time is 610 sec. without and 576 sec. with preconditioning. Here the preconditioned CG already is the better method in the sequential case. The example *crash test* has a sequential execution time of 9366 sec. without and 9992 sec. with preconditioning. The third example

Speedup: proc. vs. degree	1	2	4	8	16	32
No preconditioning	1.0	1.9	3.4	5.7	8.0	9.6
k=3	1.0	1.9	3.4	6.0	8.7	11.1
k=5	1.0	1.9	3.4	6.1	9.0	11.8
Matrix-vector product	1.0	1.9	3.5	6.8	9.6	13.2

Table 3: Speedup for various degrees

Execution times	2	4	8	16	32
No preconditioning	**7.8**	**4.33**	**2.55**	1.83	1.53
k=3	8.3	4.65	2.63	**1.80**	1.42
k=5	8.8	4.87	2.71	1.85	**1.41**

Table 4: Execution times for various degrees

has about 4 millions of nonzeros. Because of the storage requirement, a sequential run is not performed. We need at least 8 processors to store the distributed matrix. The time using 8 processors is 117 sec. without and 141.2 sec. with preconditioning. In this case, the time for the preconditioned method is increased by 20% compared with the basic CG.

In fig. 11, the relative time reduction of the preconditioned method versus the basic method on one and 128 processors is shown for the three examples. On 128 processors the preconditioned method is the faster method for all examples. This demonstrate the advantages of the preconditioner for massively parallel machines. The execution times on 128 processors are given in tab. 5.

Execution times **128 proc. in sec.**	Struct. mech. 2	Struct. mech. 3	Crash test
No preconditioning	16.7	12.1	162.7
Polynomial preconditioning; degree in brackets	13.3 (5)	11.7 (3)	136.9 (9)

Table 5: Execution times for 128 processors with and without preconditioning

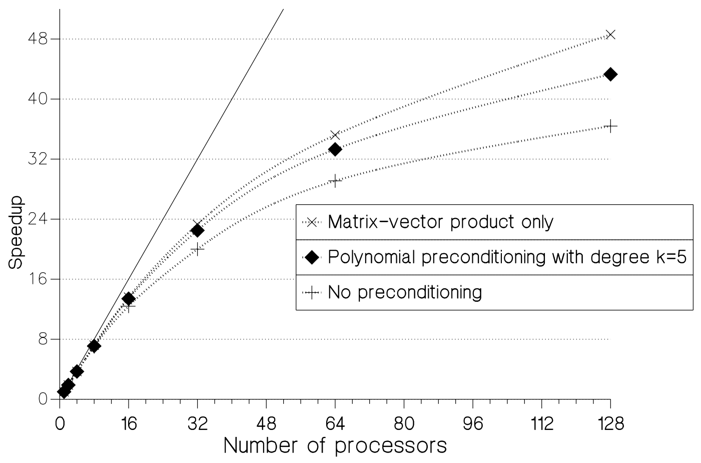

Figure 8: Speedup for *structural mechanics 2*

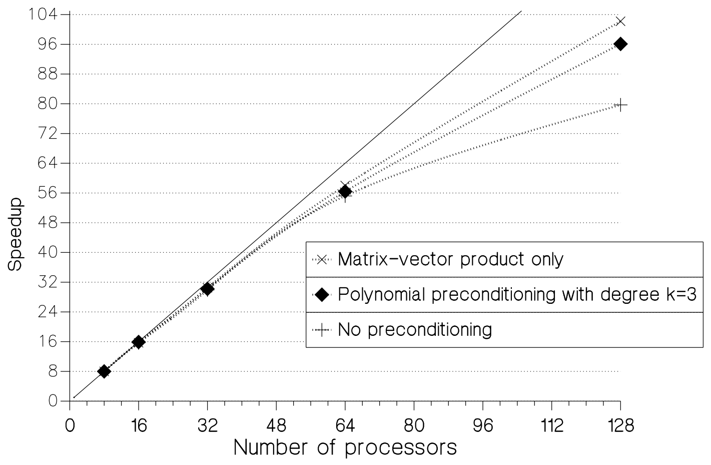

Figure 9: Speedup for *structural mechanics 3*

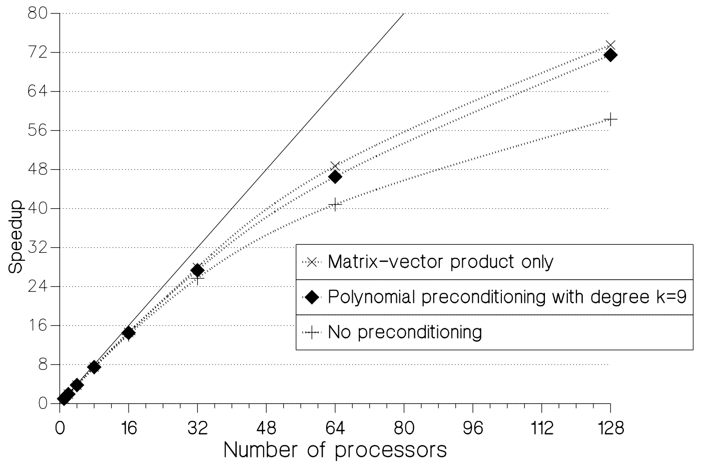

Figure 10: Speedup for *crash test*

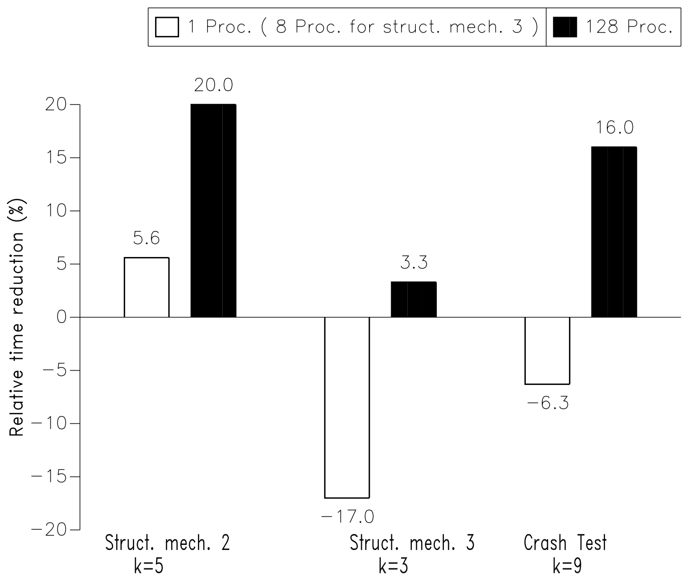

Figure 11: Relative time reduction of the preconditioned CG versus the basic CG

6 Conclusions

The method of conjugate gradients with polynomial preconditioning is an example for the fact that the best parallel algorithm is not the parallelized version of the best sequential one. The best degree of the polynomial is not fixed, but depends on several conditions, e.g., the number of processors. The performance of the preconditoned method is mainly determined by the speedup of the matrix-vector product.

We saw that increasing the degree leads to more local calculations and less global communication. By varying the degree of the polynomial, the ratio of calculation and communication can be controlled. If the speedup of the basic method decreases for a fixed degree this parameter can be increased to achieve a better efficiency. Polynomial preconditioning increases the scalability of the CG method. High degrees of the polynomial result in speedups very close to the optimum, the speedup of the matrix-vector product.

Further advantages of polynomial preconditioning are an increased stability and a higher reachable accuracy of the solution compared with the basic CG.

In addition, polynomial preconditioning is easy to integrate into a parallel basic CG on a distibuted memory machine.

To sum up, polynomial preconditioning is very attractive for massively parallel machines.

References

[1] St. Ashby *Minimax Polynomial Preconditioning for Hermitian Linear Systems*, SIAM J.Matrix Anal. Appl.,12:766-789 (1991)

[2] 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)

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

[4] A. Basermann *Datenverteilungs- und Kommunikationsmodelle für parallele CG-Verfahren zur Lösung von Gleichungssystemen mit dünnbesetzter Koeffizientenmatrix aus FE-Anwendungen*, Aachener Informatik-Berichte Nr.93-7, Graduiertenkolleg Informatik und Technik, Fachgruppe Informatik der RWTH Aachen (1993)

[5] A. Basermann *Conjugate Gradients parallelized on the Hypercube*, Int. Journal of Modern Physics C, Vol.4, No. 6:1295-1306 (1993)

[6] A. Basermann *Data Distribution and Communication Schemes for Solving Sparse Systems of Linear Equations from FE Applications by Parallel CG Methods*, Informatik Berichte TU Clausthal, Institut für Informatik, Proceedings of the Workshop on Parallel Processing in Lessach, Austria (1993)

[7] W. Bunse, A. Bunse-Gerstner *Numerische Lineare Algebra*, Teubner (1988)

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

[9] I.S. Duff *User's Guide for the Harwell-Boeing Sparse Matrix Collection, Release I*, Technical Report TR/PA/92/86, CERFACS, Toulouse Cedex (1992)

[10] G.H. Golub, Ch.F. van Loan *Matrix Computations*, 2. Auflage, John Hopkins University Press (1989)

[11] 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)

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

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

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

[15] H. Rutishauser *Theory of gradient methods*, Mitteilungen aus dem Institut für angewandte Mathematik, 8:24-49 (1959)

[16] Ch. Schelthoff *Vergleich von parallelen Verfahren zur Vorkonditionierung für die Methode der konjugierten Gradienten*, Berichte des Forschungzentrums Jülich Nr. 2913 (1994)

[17] Y. Saad *Practical Use of Polynomial Preconditioning for the Conjugate Gradient Method*, SIAM J. Sci. Stat. Comput., Vol.6, 4:865-881 (1985)