How to Improve Iterative Solvers: Parallelization and Preconditioning

Rüdiger Weiss *

Abstract

Two possibilities to accelerate the solution of linear systems are discussed and surveyed: Parallelization and preconditioning. For large linear systems resulting from technical problems usually both techniques have to be applied. There are interactions between parallelization and preconditioning that have to be considered in order to achieve efficiency. These effects will be discussed.

We also stress that convergence difficulties are often related to a non-appropriate modeling. Preconditioning could be easier on a higher level by changing the model than on the level of the linear system.

Keywords: preconditioning, parallel computers, iterative solvers, convergence

1 Introduction

To improve the speed of an iterative solver there are two possibilities:

1. use a faster computer,
2. use a faster method.

We consider both remedies. Today's supercomputers are massively parallel. We overview the main principles for programming such machines. On the algorithmic side we discuss preconditioning. Many preconditioners are recursive and, therefore, an efficient parallelization is inhibited. We draw conclusions from simultaneously using supercomputers and preconditioners.

2 Parallelization

Even if your problem has a moderate size so that you have not yet looked for supercomputers up to now, such an investigation may be a good investment for tomorrow. If we consider the near future, then parallel systems will be on your desk. Supercomputers of today are the prototypes of the commodity calculators of tomorrow. To back up this claim we look into the past and compare the Cray XMP from 1982 with the Intel Pentium processor from 1999 (see Table 1). In 1982 the Cray XMP was one of the most powerful supercomputers. It was about 1000 times faster than the Intel 80286 processor, the PC standard at that time. This was achieved not only by a cycle time that was about 10 times faster than Intel's, but also by vector processing that delivered one result in each cycle and by 4 parallel arithmetic pipes. The Intel processor still needed a mathematical

*Computing Center, Universität Karlsruhe, 76128 Karlsruhe, Germany (weiss@rz.uni-karlsruhe.de)

	1982		1999	
	Cray XMP	Intel 80286 + 80287 (math. co.)	Cray SV1	Intel Pentium III
cycle time	9.5 nsec	80 nsec	4 nsec	2 nsec
frequency	105 MHz	12.5 MHz	250 MHz	500 MHz
cycles for add	1	30	1	1
processors	2 add pipes 2 mult. pipes	1	448 vector	2 int./2 fl. p. superscalar
MFLOPS	421	0.4	1 024 000	500

Table 1: Comparison

coprocessor and 30 cycles for an add. Today the Intel Pentium III has adopted the principles of the Cray. It delivers one result each cycle and has 4 parallel processors. It is quite surprising that the Cray SV1 of 1999 has a slower cycle time than Intel's processor. This is due to the fact that in the mean time PC processors have conquered the market and that far more people are engaged in the development of PCs used in many cases as simple play stations than in the development of the high-end machines. However, the Cray gains by massive parallelism. The Intel processor of 1999 is a little bit faster than the 1982 Cray. No one would have believed this in 1982. For the future the development cycles are accelerated, so that it will probably last less than 17 years until 1 TeraFLOP is on your desk.

We follow the principles for parallelization given by Schönauer [6]. Our approach is the message passing paradigm. The user has to care about the distribution of work and data. He has to care as well for all data transfers. The advantages of message passing are portability and that the potential of distributed memory parallel computers can be exploited quite efficiently. Disadvantages are a time consuming programming and that the method is error prone. The principles for using message passing are load balancing, scalability and minimized communication.

Load balancing consists of effectively matching task requirements to the resources of a distributed computing system. The goal is to distribute load equally onto processors such that no processor idles. There is no speedup if the total work is performed by one processor and all others idle.

A program is called scalable regarding the computational time if the computation time for n operations on 1 processor is approximately equal to the computation time for $p \cdot n$ operations on p processors. Scalability guarantees that an arbitrary large problem could be solved if enough processors were available. Similarly, we define scalability regarding the memory. A program is called scalable regarding the memory if the memory requirements for n unknowns on 1 processor are approximately the memory requirements for $p \cdot n$ unknowns on each of p processors.

If the communication is blocked, then computation and communication have to be performed one after the other; see figure 1. This is the case if there is only one processor for computation and communication. Most of today's parallel computers have communication and computation processors that can operate simultaneously. By using this feature an

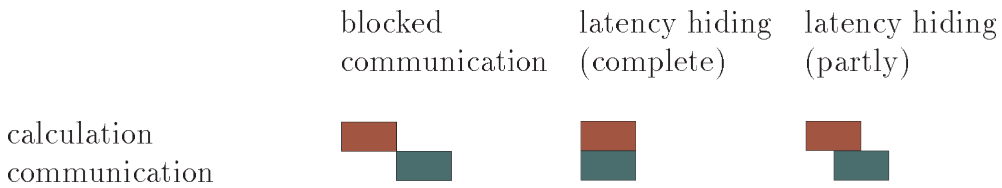

Figure 1: Latency hiding

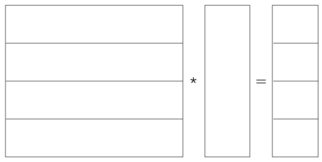

Figure 2: Matrix-vector multiplication, row-wise partition

acceleration by a factor of 2 can be achieved if the same amount of calculation and communication is done at the same time; see figure 1. This is called latency hiding. In practice usually less than 100 % can be overlaped so that latency hiding can be achieved only partly and the speedup is less than 2.

We will describe how to parallelize matrix-vector multiplications and dot products. These operations are essential for iterative solvers.

For the matrix-vector multiplication let us distribute the matrix into row blocks onto the processors like in figure 2. As rows of the matrix are multiplied by the complete vector this vector has to be stored in the memory of all processors. The algorithm is not scalable with respect to the memory. From a certain problem size on it could even be possible that the whole vector does not fit into the processor memory. If the matrix is distributed in column blocks onto the processors, then the vector can be distributed into correspondig blocks. But parts of the result vector are produced on each processor and this vector has to be stored completely. This proceeding is not scalable as well.

We describe an implementation of the matrix-vector multiplication that is scalable regarding the memory. The matrix entries are physically distributed into row-blocks onto the different processors. And, there is a second logical partition into column blocks; see figure 3. The result and the multiplicator vector are distributed accordingly. The matrix-vector multiplication is split into different steps. First the diagonal blocks are multiplied by the corresponding vector parts. All data are in the correct processor memories. In the next step the lower block diagonal including the block in the right upper corner are multiplied by the corresponding vector parts. Now the vector parts are not in the right position, but they have to be communicated in a ring shift before computation can start. In the succeeding steps all block diagonals will be multiplied by the corresponding vector parts and always the vector parts have to be sent before to the right processor. This principle is called “block-wrap-around”.

If asynchronous communication is possible, then the sending and the multiplication of the vector parts can be executed at a time. Thus, latency hiding is achieved. In order to store the incoming vector on the processors an additional buffer storage is necessary. The multiplications are performed alternatingly on the two necessary buffers, see figure

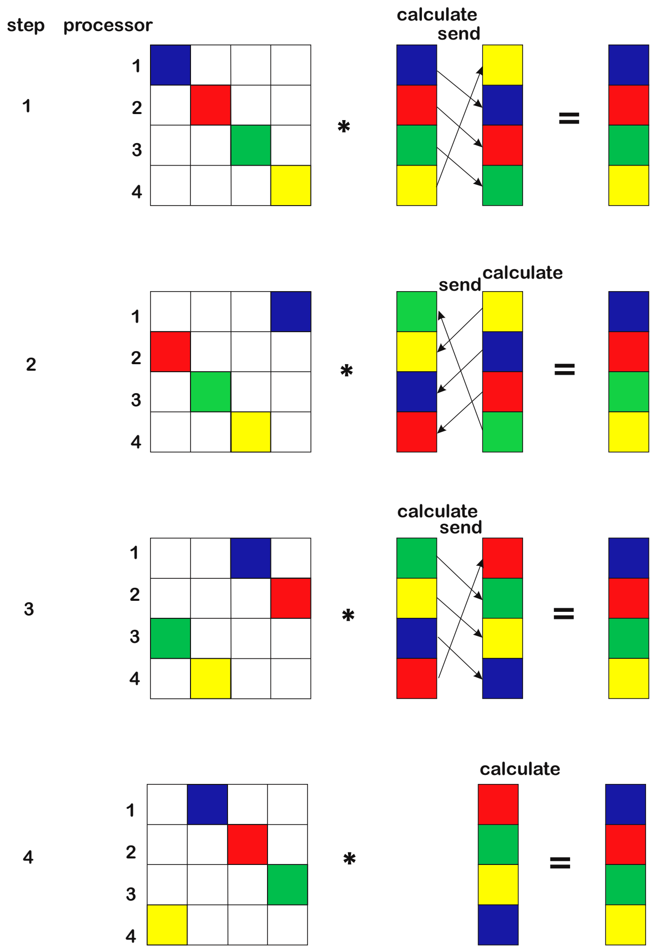

Figure 3: Matrix-vector multiplication for distributed memory computers

3. This is called alternating buffer technique.

A step of the proceeding depicted in figure 3 is omitted if all corresponding matrix blocks are empty. It is obvious that this technique is scalable regarding the memory. For full matrices it is not scalable regarding the CPU-time, because the number of steps increases with the number of unknowns. However, for sparse matrices with special patterns, where block multiplications can be omitted, scalability can be achieved.

For dot products the two vectors are distributed block-wise over the processors. Each processor calculates then a partial result that corresponds to the partial dot product of the data that are in its own memory (see figure 4). These single numbers from each processor have to be accumulated now. Each processor sends its result to the neighbor that adds the result. Every two processors have now the same partial result. This proceeding is continued in a cascaded way. At the end each processor has the complete dot product. Note that the same calculation is performed on more than one processor that would else idle. The algorithm is not scalable because if the size of the vectors is increased as the number of processors the cascade gets more stages and the communication increases. Moreover, at each stage of the cascade just a single date is sent requiring a large start-up overhead for the communication. Thus, dot products are not favorable for parallel computers with distributed memory.

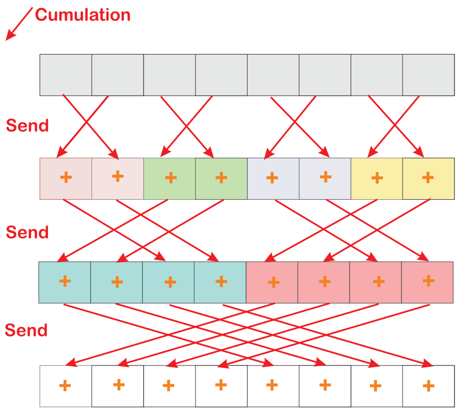

Figure 4: Dot product, communication optimized

3 Preconditioning

The idea of preconditioning is to consider a related linear system with the same solution as the original problem, that can be solved more easily. In fact today there is no efficient linear solver without any preconditioning. Instead of solving the linear system

$$Ax = b \; . \tag{1}$$

we solve the preconditioned system

$$P_L A P_R y = P_L b \; . \tag{2}$$

$P_L \in I\!R^{n\times n}$ is a non-singular left-hand preconditioning matrix and $P_R \in I\!R^{n\times n}$ is a non-singular right-hand preconditioning matrix. The solution of (1) is obtained from the solution of (2) by $x = P_R y$.

We will consider particular preconditioners and we will discuss the different phenomena in a systematic and more general form for a simple, but efficient method: Normalization or diagonal scaling. Let us consider the unbalanced equations

$$\begin{pmatrix} 10^6 & 1 \\ 0.04 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 10^6 \\ 1 \end{pmatrix} . \tag{3}$$

The size of the matrix entries may be on such a large scale because of physical reasons. If we measure the pressure in Pascal and the velocity in km/h, then there may be a range of 10^6. But there may be as well numerical reasons like non-uniform grids. If we apply a solution method that minimizes the residuals of (3), then the first equation dominates solution process. In this case you would obtain a very accurate pressure, but a completely wrong velocity. Equation (3) has the same solution as

$$\begin{pmatrix} 1 & 10^{-6} \\ 0.04 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} . \tag{4}$$

If we apply a solution method that minimizes the residuals of (4), then both equations contribute to the result: Presure and velocities are accurate. Equation (4) results from

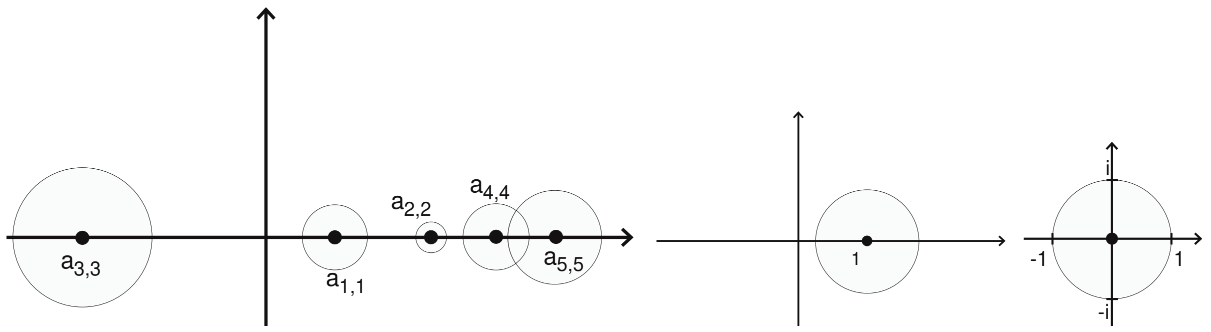

Figure 5: Eigenvalues of the original matrix and clustering

(3) by a simple multiplication by the diagonal matrix $D = diag(10^{-6}, 1)$. Diagonal scaling is appplied for physical reasons in order to balance the quantities or for numerical reasons in order to balance the solution process. It has also an acceleration effect on iterative solvers as we will see. Let us consider the eigenvalues of the system matrices in (3) and (4). For (3) we get $\approx 10^6$ and ≈ 1. For (4) we get $1 + 0.2 \cdot 10^{-3}$ and $1 - 0.2 \cdot 10^{-3}$. Thus, we have clustered the eigenvalues by fixing the diagonal elements $a_{i,i} = 1$.

This follows also theoretically from Gershgorin's theorem. The union of all discs

$$G_i := \left\{ z \in C \;\middle|\; |z - a_{i,i}| \leq \sum_{j \neq i} |a_{i,j}| \right\}$$

contains all eigenvalues of the $n \times n$ matrix $A = (a_{i,j})$. If the eigenvalues are scattered for the original matrix (see figure 5 left-hand side), then by multiplication by a diagonal matrix $D = diag(d_1, \ldots, d_n)$ with $d_i = \frac{1}{a_{i,i}}$) the eigenvalues of a diagonally dominant matrix are clustered near 1, so that $a_{i,i} = 1$ and $|z - 1| \leq \sum_{j \neq i}^{n} |a_{i,j}|$ as shown in the middle of figure 5.

Different strategies of diagonal scalings or normalization are

I. $d_i = \frac{1}{a_{i,i}}$, called Jacobi normalization.

II. $d_i = \frac{1}{\sum_{j=1}^{n} |a_{i,j}|}$. The eigenvalues are in the disk $|z| \leq 1$, figure 5 right-hand side.

III. $d_i = \frac{1}{\sqrt{\sum_{j=1}^{n} a_{i,j}^2}}$

IV. $d_i = \frac{a_{i,i}}{\sum_{j=1}^{n} a_{i,j}^2}$ minimizes the Frobenius norm of $DA - I$.

Let us apply diagonal scaling to the matrix resulting from a finite difference discretization and linearization by Newton's method of the rough model of the stationary Navier-Stokes Equations

$$\underbrace{-\Delta v}_{elliptic} + R\underbrace{(v^T \nabla)v}_{hyperbolic} - v + h = 0 \qquad (5)$$

with $R = 1$ on a $20 \times 20 \times 20$ grid. From the numerical tests in figure 6 with PRES20 [9], where O denotes no scaling, we see that scaling generates convergence. The particular type of scaling is less important. This behavior can be explained. In [8] the eigenvalue distribution of this example is investigated for a coarser grid. Let $A = C^{-1}JC$ be diagonalizable, where J is the diagonal matrix consisting of the eigenvalues. The convergence

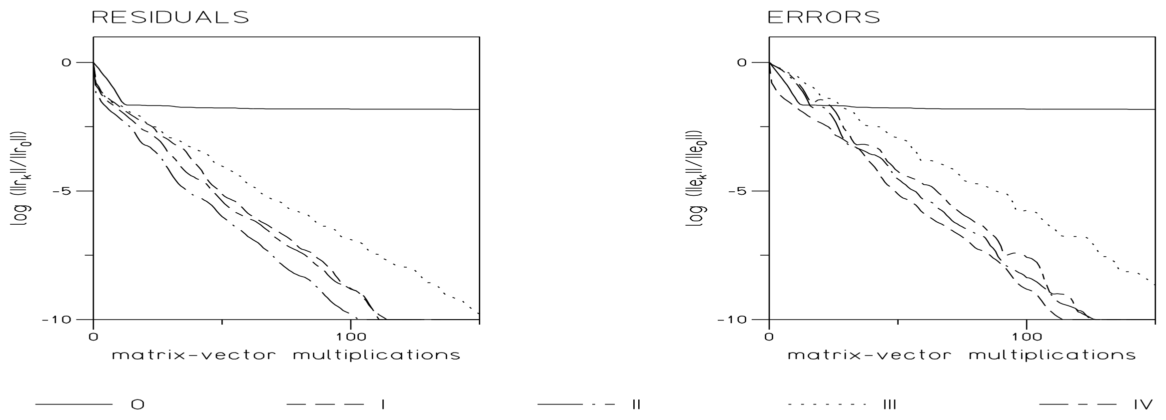

Figure 6: Effect of diagonal scaling

of Krylov subspace methods like GMRES can be characterized by

$$\|r_k\|_{any} \leq \underbrace{\kappa(C)}_{eigenvectors} \underbrace{\min_{p_k \in \Pi_k} \max_{\lambda \in \sigma(A)} |p_k(\lambda)|}_{eigenvalues} \|r_0\|_{any},$$

where r_k is the residual, Π_k are polynomials of degree k and $\sigma(A)$ is the spectrum of A. We achieve a bad convergence, if the eigenvalues are scattered or if we have non-orthogonal eigenvectors or even an eigenvector deficit. For normal matrices ($AA^T = A^T A$) $k(C) = 1$ and the convergence depends only on the eigenvalues. Thus, clustering improves convergence. Clustering also improves the convergence of the non-normal matrix related to (5) though the factor $k(C)$ could be large. In general the convergence for non-normal matrices does not dependent only on the eigenvalues. Greenbaum and Strakoš [4] show that for any given eigenvalue distribution and for any given convergence curve, there is a matrix with the given eigenvalues for which GMRES produces precisely the given convergence curve. However, normalization "normalizes" matrices in some sense as well. To get a rough idea for this effect, let us consider the normalization according to III, then

$$\|DAA^T D^T - A^T D^T DA\|_{Frobenius} \quad \leq \quad 2\|A^T D^T\|^2_{Frobenius} = 2\left(\sum_{i=1}^{n} d_i \sqrt{\sum_{j=1}^{n} a_{i,j}^2}\right)^2 \overset{III}{=} 2n^2.$$

This expression does not depend explicitely of the matrix.

Krylov subspace methods converge more and more slowly if R increases in (5). The investigations in [8] show that then the eigenvalues get more and more scattered and that the matrix becomes indefinite. Let us look for the original reasons of these difficulties. The first Laplacian term in (5) is related to an elliptic boundary value problem, while the second term with the first derivatives corresponds to a hyperbolic initial value problem. Thus, if we increase R we depart from a well-posed problem to an ill-posed problem. The origin of the problems is in the modelling, but all problems are shifted to the solution of the linear systems.

To accelerate the convergence, the physical model could be adapted: Change boundary conditions or use FOSLS (first order sytems least square) propagated by Cai, Manteuffel

technical problem
$\downarrow$ $\qquad\qquad$ $\leftarrow$ FOSLS
mathematical model
$\downarrow$
simplification of the model $\qquad$ measurements
$\searrow$ $\qquad$ $\swarrow$ $\qquad\qquad$ $\leftarrow$ upwind
$\searrow$ $\qquad$ $\swarrow$ $\qquad\qquad$ $\leftarrow$ preconditioning
numerical algorithm
$\downarrow$
program
$\downarrow$
numerical result

Figure 7: Models and errors

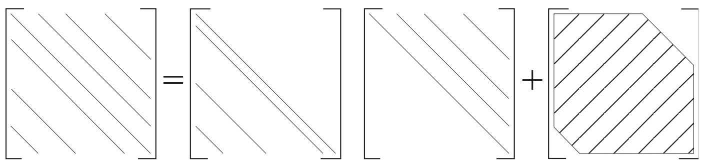

Figure 8: ILU

and McCormick [2]. On the level of the numerical method, upwind differences could be used or preconditioning. If we remember that preconditioning transforms the problem so that the solution is easier, then this transformation could also be done on a higher level; see figure 7. A "preconditoning" on a higher level would be posibly more adequate and easy.

3.1 Matrix Factorizations

One efficient preconditioning technique on sequential computers is the incomplete LU decomposition (ILU). L is a lower triangular matrix and U an upper triangular matrix. The matrix A is decomposed according to

$$A = LU + R \tag{6}$$

with $r_{k,l} = \begin{cases} 0 & if \quad (k,l) \in J \\ a_{k,l} - \sum_{j=1}^{\min(k,l)} l_{k,j} u_{j,l} & if \quad (k,l) \notin J \end{cases}$, see figure 8. J is the prescribed pattern for the factors. The pattern also can be adaptively changed according to different strategies. As preconditioned system use $L^{-1}AU^{-1}$, but never calculate the inverses of L and U. If $w = L^{-1}AU^{-1}z$ s needed, then solve

1. $Uh_1 = z$ (backward substitution)
2. $h_2 = Ah_1$ (matrix-vector multiplication)
3. $Lw = h_1$ (forward substitution)

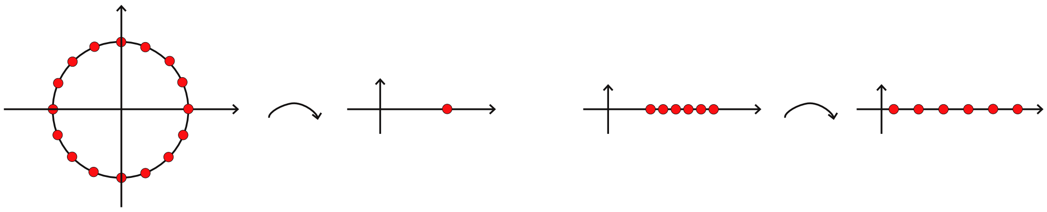

Figure 9: Normal equations

As L and U are triangular steps 1 and 3 can be easily performed by a forward-backward substitution, but the process is highly recursive, and therefore hardly suited for parallel computers. There are many modifications of ILU preconditioning and some very elaborate techniques for matrices with special patterns using efficiently supercomputers, see Fujino and Takeuchi [3]. For general matrices ILU preconditioning can be used on scalar computers, but is very slow on parallel machines.

Incomplete Cholesky decomposition (IC) is the corresponding method for symmetric matrices, see Varga [7]. Incomplete QR-Decompositions (IQR) factorize the matrix with an orthonormal Q and an upper triangular R. All these techniques are recursive.

3.2 Incomplete Inverses

Incomplete inverses try to approximate the inverse of A directly so that an additional matrix-vector multiplication has to be performed. This process is optimally parallelizable. The incomplete Frobenius inverse minimizes

$$\|AP - I\|_{Frobenius} \tag{7}$$

for a fixed pattern of P. To determine P from (7) n small independent linear systems have to be solved. This can be done in parallel. There are many variants and modifications to achieve robustness. Very promising is the variant of Huckle and Grote [5] who change adaptively the pattern of the matrix. However, this adaptive process is expensive and recursive so that the advantage of incomplete inverses may be deteriorated.

3.3 Others

Let us consider the normal equations $A^TAx = A^Tb$ and $AA^Ty = b$ as preconditioner. If the eigenvalues lie in an annular domain with the origin as center then the eigenvalues are clustered by multiplying by A^T on the real axis (see figure 9 left-hand side). In this case the convergence is accelerated. But if the eigenvalues are scattered they become even more scattered for the normal equations and convergence becomes slower (see figure 9 right-hand side). Thus the normal equations are sometimes a remedy, and sometimes not.

For Krylov subspace methods polynomial preconditioning cannot be recommended, because the original method already constructs polynomials with optimal coefficients. For very special matrices polynomial preconditioners can achieve an acceleration on massively parallel systems when dot products are avoided. But in general polynomial preconditioners lead to a slower convergence.

Multilevel methods can be considered as very fast and efficient preconditioners, but they have to be tuned to the special problem. They are not robust in the sense that parameters and strategies have to be selected very carefully and individually for each application.

Other techniques like incomplete Gauss-Jordan decompositions or iterative solvers as preconditioners seem to be not robust as well, even if there a good results for special problems.

4 Conclusions

Up to now no preconditioner is efficient, robust and parallelizable at the same time. Further research is necessary. Moreover for some techniques there is no theory up to now. Benzi and Tůma [1] recently investigated most of today's preconditioners in a very comprehensive work.

References

[1] M. Benzi and M. Tůma. A comparative study of sparse approximate inverse preconditioners. *Applied Numerical Mathematics*, 30(2–3):305–340, 1999.

[2] Z. Cai, T. A. Manteuffel, and S. F. McCormick. First-order system least squares for velocity-vorticity-pressure form of the Stokes equations, with application to linear elasticity. *Electronic Transactions on Numerical Analysis (ETNA)*, 3:150–159, 1995.

[3] S. Fujino and T. Takeuchi. ILU factorization well suited to the vector processor using a variant of the 5-point difference scheme. *Comp. Phys. Comm.*, 85(3):371–381, 1995.

[4] A. Greenbaum and Z. Strakoš. Matrices that generate the same Krylov residual space. *IMA Volumes in Math. and its Appl.*, 60:95–119, 1993.

[5] T. Huckle. Approximate sparsity patterns for the inverse of a matrix and preconditioning. *Applied Numerical Mathematics*, 30(2–3):291–303, 1999.

[6] W. Schönauer. Scientific supercomputing. [Online.] Version 1.0. [Karlsruhe, Germany], 1998. <http://www.uni-karlsruhe.de/Uni/RZ/Personen/rz03/book/>.

[7] R. S. Varga. Factorization and normalized iterative methods. In R. Langer, editor, *Boundary Problems in Differential Equations*, pages 121–142. University of Wisconsin Press, Englewood Cliffs, 1960.

[8] R. Weiss. Convergence behavior of generalized conjugate gradient methods. Report 43/90, University of Karlsruhe, Computing Center, 76128 Karlsruhe, Germany, 1990.

[9] R. Weiss. *Parameter-Free Iterative Linear Solvers*. Mathematical Research, vol. 97. Akademie Verlag, Berlin, 1996.