Numerical experiments to optimize the use of (I)LU preconditioning in the iterative linear solver package LINSOL

Willi Schönauer and Hartmut Häfner

Rechenzentrum der Universität Karlsruhe
D-76128 Karlsruhe, Germany [1]

Dedicated to the memory of Rüdiger Weiss

Abstract

In the iterative linear solver package LINSOL several generalized conjugate gradient (CG) methods (or, briefly, CG-type methods) with quite different properties are implemented. With these methods polyalgorithms with automatic method switching are constructed. The "emergency exit" that is taken in the worst case is the ATPRES method (which is very robust, but very slow). In this paper we investigate, if (I)LU preconditioning were a better emergency exit and how the drop tolerance for small elements in ILU affects the convergence behavior. The answer will be: it depends.

Key words: sparse linear systems, (I)LU preconditioning, software, LINSOL, FIDISOL, automatic method switching, generalized conjugate gradient (CG) methods, polyalgorithms

1 Introduction

The LINSOL program package has been designed originally as the iterative linear solver integrated into the PDE black-box solver FIDISOL, see chapter 17 in [1]. Presently, in LINSOL, 14 generalized conjugate gradient (CG) methods are implemented, and LINSOL has been efficiently parallelized [2,3,4]. These different generalized CG methods have quite different properties, see [5,6]. Therefore we have developed "polyalgorithms" with automatic method

[1] Email of first author: schoenauer@rz.uni-karlsruhe.de.

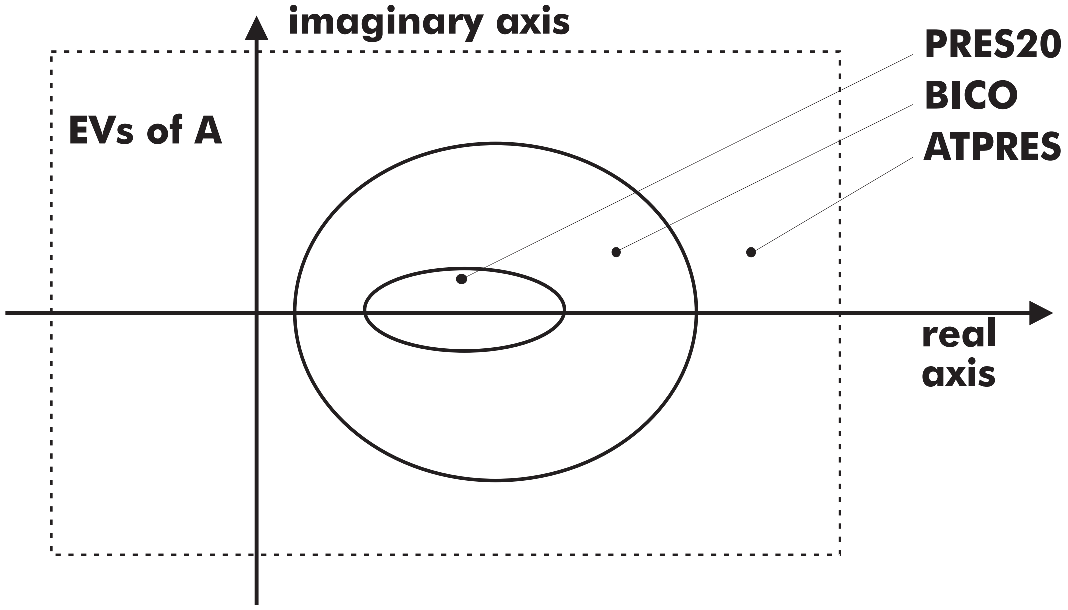

Fig. 1. Range of optimal application for the methods PRES20, BICO, ATPRES depending on the eigenvalues of the matrix A of the linear system.

switching. A typical representative is the following, see section 16.4 in [6], where the methods used are also described:

Start with the PRES20 method. This is an ORTHORES(5) method using five preceding residuals and being restarted every 20 iterations from the smoothed sequence (based on experimentally determined very robust parameters), see below in chapter 4. It is for sufficient diagonal dominance the fastest method. Unfortunately, it is also the least robust method, which loses soon its efficiency if the diagonal dominance is less pronounced. Every 20 iterations, i.e. at each restart, we check if the L_2-norm of the residual has been reduced at least by a factor of 0.5. It should be mentioned that we normalize the linear system when starting LINSOL so that the absolute row sums are equal to one. If the reduction factor is > 0.5, we switch to the BICO method.

BICO is the classical bi-conjugate gradient method, but controlled by the smoothed sequence. It is medium fast and medium robust. Because of the plateau behavior of BICO ("falling down a cliff" after a stagnation period of the residual norm), we check after 500 iterations, i.e. after 1000 matrix-vector multiplications (MVMs) if the residual has been reduced at least by a factor of 0.1. If the reduction is worse, i.e. if the factor is > 0.1, we switch to ATPRES.

ATPRES is Craig's method (also called CGNE) with control by the smoothed sequence. It is extremely robust, i.e. it converges even in very critical cases, but it is also extremely slow. We have selected ATPRES as the "emergency exit" of our polyalgorithms. It should be mentioned that there are also other polyalgorithms, including some that try to return to the previous method.

Fig. 1 shows how the ranges of optimal application for the three methods mentioned above depend on the eigenvalues of the matrix A of the linear system to be solved. Remember that LINSOL is part of black-box PDE solvers (FIDISOL, CADSOL, VECFEM [7], FDEM [8,9,10], where the last two operate on unstructured grids). The properties of the matrix A depend on the PDEs the user submits to the black box, e.g. the Navier-Stokes equations with low or high Reynolds number. They change when going from a coarse to a very fine grid. Thus the properties of the matrix A that results from the discretization are completely arbitrary. This is quite different from the case of a PDE solver for a special (type of) PDE. In the case of the FDEM program package we have additionally the choice to select the consistency order 2, 4 or 6, which also changes the properties of the matrix (higher order means more nonzero entries in a row), and we have implemented a self-adaptation of mesh size and consistency order so that order and mesh size are not fixed in advance. This characterizes the extreme situations in which LINSOL must solve efficiently the resulting linear systems.

It is obvious that if the first two methods of the polyalgorithm (PRES20, BICO) converge as requested, there is no need for improvement. However, if LINSOL finally switches to ATPRES, the solution procedure may become very inefficient. Therefore we posed the question, whether we could improve the action taken in case of an emergency exit by preconditioning. But this preconditioning has to be very robust.

2 Preconditioning

If one consults recent textbooks on iterative linear solvers, e.g. those of Saad [11], Greenbaum [12], Dongarra et al. [13], and Meurant [14], large sections are devoted to preconditioning. The goal is always that the eigenvalues of the preconditioned system are clustered in the sense of Fig. 1 so that an iterative solution, typically by a CG-type method, becomes more efficient or, perhaps, possible at all. It is clear that in the LINSOL environment only very robust preconditioners can be used. Under these conditions (incomplete) LU or factorized (incomplete) AI (approximate inverse) preconditioners are the favorite ones. In [15] the equivalence of both types is proved under certain conditions. In [16] Benzi and Tuma compare several sparse approximate inverse preconditioners.

As we want to design a code for distributed memory parallel computers, factorized (I)AI seems to be the ideal method because it requires only MVMs once the factorization is done. However, as we want an extremely robust preconditioner, we must include the option of a full factorized inverse which means a full lower and a full upper triangular matrix. It should be mentioned that for factorized IAI the structure of the "significant" nonzero elements of L^{-1} and U^{-1} (where $A^{-1} = U^{-1}L^{-1}$) is much easier to determine than for the inverse A^{-1} itself or its approximation. Our decision, for reasons of storage and consequently for computation, was to implement (I)LU as preconditioner because for a large sparse band matrix and diagonal pivots, the full factors L and U have the same bandwidth as the matrix A itself. But as we admit full pivoting, the bandwidth of one of them may get doubled, depending on whether the zeros are created below or above the main diagonal. If the zeros in the Gauss elimination are created below the main diagonal we have (I)LU, if they are created above we have (I)UL. For reasons of data structure we use the latter form. Then U is lower triangular and has up to the double bandwidth of L. The details of the data structure for (I)UL in LINSOL are presented in [17].

The next question is: with which basic CG-type method should we combine (I)UL? The most expensive part of (I)UL is the factorization $A = UL$. Then in each iteration step forward elimination with U and backward substitution with L must be done. On a parallel computer the factorization can be efficiently parallelized if the appropriate data structures are selected. However, it is not possible to parallelize efficiently the recursive forward elimination and the backward substitution. PRES20 needs one forward elimination and one backward substitution with the factors of the matrix A per iteration step whereas BICO needs one with the factors of the matrix A and another with the factors of the matrix A^T. If the data structures are optimal for A, they are just the contrary for A^T, and a transposed structure is needed for efficient execution. Moreover, as PRES20 is the fastest method for clustered eigenvalues of A, we can expect that it converges very fast if a sufficiently "good" (I)UL is used. Another candidate for the combination with (I)UL is the widely used Bi-CGSTAB of van der Vorst [18] which is based on the biconjugate gradient method, but avoids A^T while needing two applications of A per step. Here, this means two forward eliminations and two backward substitutions per iteration step. In this paper we report about (I)UL for PRES20. In [17] also some results for (I)UL for Bi-CGSTAB are presented.

In many papers about preconditioning only the number of iteration steps needed for a certain residual reduction is reported. This is generally of no relevance because the computer time is the important measure. As we want to solve extremely large problems on distributed memory parallel computers, the time depends significantly or even exclusively on the data structures that are used. This holds especially for the implementation of (I)UL. So all observations and conclusions that we will draw from our numerical experiments hold <u>*only*</u> for our (present) implementation and cannot be generalized to other implementations without careful investigations. Inefficient parallelization may lead to different conclusions.

3 The expectations

Here we report about our expectations that we had before we started our investigations. The reader can compare them — as we did — with the results obtained. For UL-preconditioning the factorization $A = UL$ is surely the most expensive part. As we will see in the next chapter, this means that the solution with PRES20 is obtained in just one iteration step. If we use IUL, we control the dropping of small elements by a relative threshold value `threshold2`, which means that elements of U and L generated in the factorization process are discarded if in absolute value smaller than `threshold2` times the largest absolute value of an element of the normalized matrix A. Normalization is usually enforcing absolute row sums equal to one.

There is another value `threshold1` for dropping small elements when the matrix A is copied into the working array. This parameter turned out to be rather "dangerous". If we dropped even very small elements, the IUL preconditioning failed, even when the `threshold2` was set to 0, which means full UL. So, in this paper, we investigate only the influence of `threshold2`, while `threshold1` was set to 0 (drop nothing).

What is the expected influence of `threshold2`? If `threshold2` $= 0$, this means full UL factorization, and thus solution in one iteration step. Small values of `threshold2` means dropping relatively few elements. The factorization is executed in a buffer window with unpacked rows and with full pivoting. Only if a row becomes inactive and is copied into the U and L arrays, the filtering by `threshold2` takes place, i.e. small elements are not copied. So we admit restricted fill-in, determined only by the value of `threshold2` and not by a prescribed non-zero pattern. This procedure means also that the pure arithmetic computation time is the same for full UL and for IUL. However, to sort U and L needs less time for smaller U and L. The I (of incomplete) clearly has also an effect on forward elimination and backward substitution: for larger values of `threshold2` U and L are sparser (within the generated band) and forward elimination and backward substitution are less expensive than for smaller values of `threshold2`. So we may ask: is there an optimum value of `threshold2`, so that the total computation time is minimized? If we drop more elements (larger `threshold2`) we expect more iterations, but with sparser matrices U and L. Or is by our way of implementation of `threshold2` full UL always optimal?

Another question was: would it be sufficient to use the recursive preconditioning steps, i.e. the forward elimination and backward substitution, only every jth step (instead of every step)? When we saw the results of our first experiments, we could immediately answer this question with "no". We use the preconditioning only in extremely critical cases. And in these, PRES20 does not converge at all, but immediately stagnates. So unpreconditioned steps are only a waste of time.

Another important factor for the solution of linear systems with large sparse matrices is storage. A larger value of `threshold2` means dropping more elements, and thus less storage is needed for U and L. We will see later in our examples how the storage for U and L changes with `threshold2`. Reserving sufficient storage for U and L is an important computational parameter that is not known in advance. If the reserved storage space is too small, the program stops and reports how far it has proceeded, so that one can estimate how large the storage space should be. After a successful run, the size of the storage used is printed out.

4 Some formulae

The formulae of the preferred methods used in our LINSOL program package have been summarized in [19]. We want to solve the linear system

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

The residual for an iterate x_k of some iterative method is defined by

$$r_k = Ax_k - b. \tag{2}$$

The system (1) is assumed to be normalized (usually by absolute row sums equal to 1). The basic approach for the PRES20 method is the following: we define a pseudo-residual $\overline{r}_{k+1}$ that is related to the true residual r_{k+1} by

$$r_{k+1} = \varphi_k \overline{r}_{k+1}, \tag{3}$$

i.e., the true residual and the pseudo-residual have the same direction. The new pseudo-residual is a combination of p preceding true residuals:

$$\overline{r}_{k+1} = Ar_k + \alpha_{1,k} r_k + \alpha_{2,k} r_{k-1} + \ldots + \alpha_{p,k} r_{k-p+i}. \tag{4}$$

For the choice

$$\varphi_k = \frac{1}{\alpha_{1,k} + \ldots + \alpha_{p,k}} \tag{5}$$

we get the new solution

$$x_{k+1} = \varphi_k(r_k + \alpha_{1,k} x_k + \ldots + \alpha_{p,k} x_{k-p+1}). \tag{6}$$

The coefficients $\alpha_{i,k}$ are chosen so that $\overline{r}_{k+1}^T \overline{r}_{k+1} = \min$, which means that

$$\alpha_{i,k} = -\frac{r_{k-i+1}^T A r_k}{r_{k-i+1}^T r_{k-i+1}}. \tag{7}$$

The consequence of the minimum condition is that

$$r_{k+1}^T r_{k-i+1} = 0, \qquad i = 1, 2, \ldots, p, \tag{8}$$

i.e., the new residual is orthogonal to the p preceding residuals. Hence the name ORTHORES(p) [20].

Because the residual r_{k+1} may oscillate heavily, we compute in parallel to the original sequence a smoothed residual sequence

$$s_{k+1} = \alpha_k s_k + (1 - \alpha_k) r_{k+1}. \tag{9}$$

Here, α_k is also chosen so that s_{k+1} has minimum length, i.e.,

$$\alpha_k = \frac{r_{k+1}^T (s_k - r_{k+1})}{(s_k - r_{k+1})^T (s_k - r_{k+1})}. \tag{10}$$

¿From the definition $s_{k+1} = A x_{k+1,smoothed} - b$ of the residual it follows from (9) that the new smoothed approximate solution is

$$x_{k+1,smoothed} = \alpha_k x_{k,smoothed} + (1 - \alpha_k) x_{k+1}. \tag{11}$$

PRES20 takes $p = 5$ and makes a restart every 20 iterations from the smoothed sequence. At the beginning we select x_0, compute r_0 from (2), and set $s_0 = r_0$. At a restart, $r_k = s_k$ and $x_k = x_{k,smoothed}$. At the beginning and after a restart (4) is built up with 1, then 2, etc., preceding residuals until $p = 5$ is reached, and then we keep $p = 5$ preceding residuals until the next restart. The essential cost is due to the matrix-vector multiplication (MVM) Ar_k. The parameters $p = 5$ and the restart rate at 20 iterations have been optimized by "numerical engineering" and are very robust, i.e., they hold for quite different types of matrices. The advantage of PRES20 is that it needs only one MVM per iteration step and that it is the fastest method for sufficient diagonal dominance. The disadvantage is that it may stagnate if the diagonal dominance is less pronounced or even lost.

The reason why we chose to combine (I)UL preconditioning with PRES20 is two-fold: PRES20 needs only one forward elimination/ backward substitution per iteration step, and if the eigenvalues of the preconditioned system are sufficiently clustered, PRES20 is the fastest method.

For preconditioning we replace in the basic approach (4) the term Ar_k by APr_k where P is a preconditioning matrix. This is called "preconditioning from the right". This means we have replaced the "search direction" r_k by a "better" search direction Pr_k. What would be the best search direction? If $APr_k = r_k$, this term cancels with $\alpha_{1,k} r_k$ for $\alpha_{1,k} = -1$ in (4), and with $\alpha_{i,k} = 0$ for $i > 1$ we get $\overline{r}_{k+1} = 0$, and thus $r_{k+1} = 0$, $x_{k+1} = x$. The relation $APr_k = r_k$ holds for $P = A^{-1}$. If we use the factorization $A = UL$, we get $P = A^{-1} = L^{-1}U^{-1}$ and thus

$$APr_k = \underbrace{A\,\underbrace{L^{-1}\,\underbrace{U^{-1} r_k}_{forward\ elimination}}_{backward\ substitution}}_{MVM} = Az_k. \tag{12}$$

So we have a forward elimination with U and a backward substitution with L that result in the vector z_k; finally we need the MVM Az_k. In the formula (6) for x_{k+1} we must replace r_k by $z_k = L^{-1}U^{-1}r_k$. This is for full UL. In case of IUL we have

$$A = U^* L^* + R, \tag{13}$$

where U^* and L^* are approximations for U and L that result from the dropping of small elements. The remainder R is neglected. Then all statements and formulae that we gave for U and L now hold for U^* and L^*, except that we do not get the solution in one step but hopefully in only a few iterations, depending on the influence of R. In order to better explain the meaning of (12) we have with $P^* = L^{*-1}U^{*-1}$:

$$\begin{array}{lll} \text{solve} \quad U^* w_k = r_k & \Rightarrow & w_k = U^{*-1} r_k, \\ \text{solve} \quad L^* w_k = z_k^* & \Rightarrow & z_k^* = L^{*-1} w_k, \\ \text{compute} \quad AP_k^* r_k = Az_k^*. & & \end{array} \tag{14}$$

For full UL we have $z_k^* = z_k$. As mentioned above, it will be very interesting, which part of U and L must be retained in U^* and L^* so that we get optimal total solution time.

As mentioned above, (I)UL with `threshold2` for dropping small elements means possibly considerable fill-in. We characterize fill-in by two values that we explain in the following: the Gauss factor GF and the fill-in factor FIF. For this purpose we must first explain how LINSOL acts as integrated part of our PDE black-box solver FDEM [8,9,10]. FDEM solves nonlinear systems of elliptic and parabolic PDEs of type

$$Pu \equiv P(t, x, y, z, u, u_t, u_x, u_y, u_z, u_{xx}, u_{yy}, u_{zz}, u_{xy}, u_{xz}, u_{yz}) = 0. \tag{15}$$

If we have a system of m PDEs, u and Pu have m components. At the boundaries we admit the same operator type. The elements of the large sparse matrix that we call $\tilde{A}$ are generated as products of coefficients of difference formulae and of elements of Jacobian matrices $\partial Pu/\partial u_{\nu,\mu}$, e.g. $\partial Pu/\partial u_{xz}$. If u_{xz} does not appear in Pu, then $\partial Pu/\partial u_{xz} = 0$, and at the corresponding position in the matrix $\tilde{A}$ a zero is "computed". On a pipelined computer this is much faster than to avoid the zeros by if-statements. The matrix $\tilde{A}$ is stored as packed rows, i.e. the "nonzero" elements and their column indices in the row are stored. "Natural" zeros that do not appear in the difference scheme are not stored. However, the "computed" zeros are included in the "nonzero" elements. This matrix $\tilde{A}$ is handed over to LINSOL when LINSOL is called. On a parallel computer with p processors each processor gets a block of $\approx n/p$ rows, if n denotes the dimension of $\tilde{A}$.

The first step of LINSOL is to "purify" $\tilde{A}$ from the computed zeros; this results in the matrix $\hat{A}$. For quite simple (systems of) PDEs there may be 60 - 70 % computed zeros. If all terms of the operator (15) are present in the PDE, there are no computed zeros. So, $\tilde{A}$ and $\hat{A}$ may differ significantly in the number of elements. The next step in LINSOL consists in normalizing $\hat{A}$; this results in the matrix A that is used in the iterative solution.

If for any matrix M we denote by $\sharp M$ the number of elements of M, we define the Gauss factor GF by

$$\mathrm{GF} := \max_{ip=1(1)p} \frac{\sharp(U+L)_{ip}}{\sharp\mathrm{LMAT}_{ip}}, \tag{16}$$

where ip is the processor number, p is the number of processors, and LMAT is the array that is reserved for storing $\tilde{A}$; its size is usually close to the size of $\tilde{A}$. The factor GF expresses how many times the array reserved for $U + L$ must be larger than the one for the matrix $\tilde{A}$. As one does not know GF in advance, an estimated value of GF must be used, and the program reports afterwards the value of GF that was needed. If the trial value was too small, the program stops and reports at which row of A the factorization ran out of space. Clearly, the introduction of GF was motivated by practical implementation considerations.

(I)UL is executed with the matrix A that has the same size as $\hat{A}$. In order to see the "true" fill-in, as one would intuitively define it, we introduced also the following fill-in factor FIF:

$$\mathrm{FIF} := \frac{\sharp(U+L)}{\sharp A}. \tag{17}$$

FIF tells us, how many times the storage for $U + L$ must be larger than the storage for A. As we introduced FIF when some of our experiments were done already, its value is not available for all the numerical examples presented here, as some of the computations were very expensive.

5 Numerical examples

5.1 2-D examples

As LINSOL is an integrated part of our black-box PDE solver FDEM [8,9,10] we use matrices generated by FDEM for the following 2-D nonlinear elliptic system of three PDEs:

$$\begin{aligned} u_{xx} + u_{yy} + \omega_y - f_1 &= 0, \\ v_{xx} + v_{yy} - \omega_x - f_2 &= 0, \\ u\omega_x + v\omega_y - \tfrac{1}{R}(\omega_{xx} + \omega_{yy}) - f_3 &= 0, \end{aligned} \tag{18}$$

with the boundary conditions

$$u - g_1 = 0, \quad v - g_2 = 0, \quad \omega + u_y - v_x - g_3 = 0 \tag{19}$$

on the 2-D rectangular domain $x \in [-2, 2]$, $y \in [-2, 2]$. The forcing functions f_i and g_i are selected so that

$$\bar{u}(x, y) = \bar{v}(x, y) = \bar{\omega}(x, y) = \tfrac{1}{50}(x^6 - y^6 + x^2 y - y^4 + 3xy - x) \tag{20}$$

is the exact solution. This is also our starting point for the Newton iteration. Without the forcing functions f_i and g_i the system (18), (19) represents the Navier-Stokes equations with Reynolds number R. A large value of R makes the resulting linear system critical; in physical applications there is possibly no (stable) steady solution for sufficiently large R. In our case we have the "artificial" solution (20), and it is not known how such a system will behave for large R.

We use a regular triangular mesh (linear triangles) on which we use a 4th order FDM. As we include in FDEM an error estimate by a polynomial of order 6, we get a "nearly" exact error estimate to control the solution.

What is the "scale" we use to judge the method? The only useful scale is time, usually CPU time in sec. The number of iterations is not a useful scale. However, we have made the experience that by the internal control algorithms of FDEM the stopping criteria for the first or the second Newton iteration or for the error computation are quite different. For example, a better solution of the linear system in the first Newton step may avoid a second Newton step. What we want is a *solution of the problem*, not the solution of an isolated linear system. Therefore we take as our scale the solution time for the whole problem. This includes the generation of the difference formulae, for each complete Newton step the generation of the matrix (but not for modified Newton steps), the computation of the right-hand side for each Newton step and the computation of the right-hand side for the error computation.

The experiments were run on an IBM RS/6000 SP with 120 MHz Power2SC processors (with a theoretical peak of 480 MFLOPS per processor). We used 8 processors for the 2-D examples. FDEM is efficiently parallelized, including the (I)UL in LINSOL.

The grid had 183×183 grid lines, which results in 33489 nodes with 66248 triangular elements and $n = 100467$ unknowns. The maximum distance of a nonzero element from the main diagonal is 4115, so the nonzero elements are within a band of width 8229.

Which value of R must be used to get a sufficiently "critical" linear system, so that BICO fails and only ATPRES still (slowly) converges? The (naive) expectation was that with full UL we would get a solution for any value of R. However, while for $R = 100$ we obtained a solution, for $R = 1000$ the Newton iteration diverged. Further experiments showed that between $R = 562$ and $R = 564$ even the damped Newton iteration ceases to converge. So we selected $R = 500$ for the numerical experiments. For this value the Newton iteration converges without relaxation factor.

Remark: As we have designed our PDEs so that the 6th order polynomial (20) is the solution, there must be a solution for every value of R. With a 6th order FDM we "confirmed" the solution; the residual and the error estimate were in the region of rounding errors. If we started with an initial solution that was disturbed by 1 % the solution was reproduced in 5 Newton steps. For 10 % disturbance the Newton iteration nearly stagnated, and the computing time exceeded 4 h.

We also investigated how well CG-type methods behaved for small values of R. PRES20 and BICO stagnated even for very small R. Only BiCGSTAB converged up to $R = 100$ with the following timings:

R	MVMs	CPU sec
0,1	1665	181
1	1379	174
10	1958	187
100	12523	429

Here MVM is the number of matrix-vector multiplications.

Our emergency exit of the polyalgorithm, the robust and slow ATPRES, converged for $R = 500$ with 181 078 MVMs, i.e. 90 539 iterations in 4266 sec (*approx* 1.2 h). Here the number of iterations is close to the dimension $n = 100467$ of A. The question was now, if we could get with UL or IUL better timings than 4266 sec.

Table 1 shows the tests with different values of the relative drop tolerance `threshold2`, the value 0 (drop nothing) is full UL preconditioning. For UL two iteration steps reduce the residual of the linear system below the requested stopping criterion for the normalized residual in the L_2 norm. Then the maximum norm of the original system is computed with another MVM. Because the requested stopping criterion is then not fulfilled, another iteration step is executed that brings the residual below the L_2 stopping criterion and another MVM checks the maximum norm, which is then below 10^{-16}, so that the iteration stops. So we have 3 iterations and 5 MVMs for the computation of the Newton correction, we need only one Newton step. The same sequence of operations holds for the computation of the error estimate, so that we have finally recorded 10 MVMs. We would expect for full UL 2 iterations which would result in 4 MVMs (including the "control" MVM). However, the linear system is so critical that by the influence of rounding errors a single iteration is not accurate enough for the Newton iteration or the error computation. The additional iterations are ultimately the correction of the rounding errors.

The Gauss factor GF for full UL is 51.7, which means that we need approximately the 51.7-fold storage of $\tilde{A}$ for $U + L$, and the fill-in factor FIF is 104.9, which means that $U + L$ need the 104.9-fold storage of A. This means also that in $\tilde{A}$ there are roughly 50 % computed zeros. In [11], section 10.4.4, Saad recommends for critical matrices a fill-in parameter of 20, i.e. the 20-fold storage of the original matrix. Let us see if it will be better to drop elements using a value of `threshold2` > 0 and thus to work with a lower value of FIF.

If we select `threshold2` $= 10^{-6}$, see the second row of Table 1, GF goes down from 51.7 to 33.6 (FIF is not recorded), the storage for $U + L$ goes down from

Table 1
Tests with `threshold2` for (I)UL, $R = 500$.

`threshold2`	GF	FIF	$\sharp(U+L)$ in Mwords	MVMs	total time in seconds	
0(UL)	51.7	104.9	29.2	10	3168	
10^{-6}	33.6		18.9	12	2811	
$2 \cdot 10^{-6}$	30.1		17.0	12	2802	
$2.5 \cdot 10^{-6}$	29.1		16.4	11	2732	
$3 \cdot 10^{-6}$	28.2	46.4	15.9	10	2715	*optimal*
$4 \cdot 10^{-6}$	26.7		15.1	12	2730	
$5 \cdot 10^{-6}$	25.6		14.4	13	2742	
$7 \cdot 10^{-6}$	24.0		13.5	13	2778	
10^{-5}	22.2		12.5	14	2784	comparison
$3 \cdot 10^{-5}$	17.2		9.7	17	2808	
$5 \cdot 10^{-5}$	15.1	19.5	8.5	26	3124	
10^{-4}	12.4	14.9	6.9	49	3957	
$3 \cdot 10^{-4}$	8.8		4.9	269	11594	($\approx$ 3.2 h)
				ATPRES	4266	robust, slow

29.2 to 18.9 Mwords and the total time reduces from 3168 sec to 2811 sec. As mentioned above, for UL and IUL the number of arithmetic operations for the pure factorization is the same, but as the IUL matrices are smaller, there are less data transfer operations for storing and, above all, for sorting U and L. The pure factorization for full UL and IUL with `threshold2` $= 10^{-6}$ requires 912 and 888 sec, respectively, and the sorting into the data structure for forward elimination and backward substitution takes 1335 and 998 sec, respectively (more than the factorization time!). For IUL we have 12 MVMs resulting from a total of 8 iterations (compared to 6 for UL) and 4 "control" MVMs. However, each preconditioned iteration step is cheaper than that of UL, so that finally a shorter total time results.

If we increase `threshold2` we get for `threshold2` $= 3 \cdot 10^{-6}$ the smallest total time. For this example GF is reduced to 55 % and FIF to 44 % of that of full UL. Although there are now two additional preconditioned iteration steps, we get a shorter total time because of less communication. This shows the essential influence of the communication operations in a parallel code. The efficiency of these operations depends essentially on the data structures.

For `threshold2` $= 10^{-5}$ (row marked by "comparison") we need only 2.5 % more computation time, but we save 21% of the storage for $U + L$. So we can sacrifice computation time for saving storage space. For larger values of `threshold2` the total time soon increases and surpasses the time of ATPRES.

Since the computations shown in Table 1 were done, we improved the code by improving the selection of nodes for the difference formulae and the elimination of the computed zeros from $\tilde{A}$ to $\hat{A}$. With the new code the time for the full UL (first row in Table 1) reduces from 3168 sec to 2715 sec, so that the total time required by the new code is roughly 14 % less. With the new code the estimate of the maximum relative error is $0.31 \cdot 10^{-4}$, and the exact error is $0.23 \cdot 10^{-4}$.

5.2 3-D examples

In 3-D we use again the velocity vorticity formulation of the Navier-Stokes equations with forcing terms f_i and g_i. This yields now a system of six PDEs:

$$\begin{aligned} u_{xx} + u_{yy} + u_{zz} - \eta_z + \zeta_y - f_1 = 0, \\ \text{similar equations for } v \text{ and } w, \\ -\xi u_x - \eta u_y - \zeta u_z + u\xi_x + v\xi_y + w\xi_z - \\ -\tfrac{1}{R}(\xi_{xx} + \xi_{yy} + \xi_{zz}) - f_4 = 0, \\ \text{similar equations for } \eta \text{ and } \zeta. \end{aligned} \tag{21}$$

on $x \in [-4, 4]$, $y \in [-0.5, 0.5]$, $z \in [-0.5, 0.5]$. The boundary conditions are of the type

$$u - g_1 = 0, \quad \ldots, \quad \xi - w_y + v_z - g_4 = 0, \quad \text{etc.} \tag{22}$$

The forcing functions f_i and g_i are determined so that for all components the exact solution is

$$\begin{aligned} \bar{u}(x, y, z) = \cdots = \bar{\zeta}(x, y, z) = \tfrac{1}{4000}\Big(x^6 - y^6 + z^6 + x^2yz. \\ - y^4 + z^4 + 3xy - x + z\Big). \end{aligned} \tag{23}$$

Again a large value of the Reynolds number R makes the resulting linear system very critical.

On the $8 \times 1 \times 1$ domain we use a regular tetrahedral 3-D mesh and a 4th order FDM on that grid. The grid is $104 \times 13 \times 13$ which gives 17'576 nodes, 74'160

Table 2
Solution parameters for different values of the Reynolds number R.

	full UL for PRES20				ATPRES	
R	GF	$\sharp(U+L)$ in Mwords	MVMs	total time in seconds	total time in seconds	MVMs
1	23.5	16.2	4	2014	3416	174 846
10	23.5	16.2	4	2014		
10^2	23.7	16.4	4	1995	6501	350 478
$\rightarrow 10^3$	23.7	16.4	4	2013	5875	300 002
10^4	23.8	16.4	4	2204	8678	450 002
10^5	23.8	16.4	4	2136		
10^6	23.8	16.5	4	2117		

elements (tetrahedrons) and 105'456 unknowns. The half-bandwidth is 4547, i.e. the elements are within a band of 9093 elements. The used number of grid lines is too coarse for an accurate solution of the polynomial (23) on this $8\times1\times1$ domain. However, finer grids in 3-D domains are soon extremely expensive. As our purpose is to investigate the behavior of the iterative linear solver, the accuracy of the solution is of no importance. We now use 32 processors of our IBM SP.

Which value of the Reynolds number R should we take to make the system sufficiently critical (note that our polynomial solution has no physical meaning)? Table 2 shows numerical experiments with different values of R with full UL preconditioning for PRES20 and for some of these values also with the unpreconditioned ATPRES. In contrast to the 2-D example we get solutions for all tested values of R, and for full LU preconditioning there is no great difference in the computational parameters. For each the computation of the Newton correction and the computation of the error, one iteration step and one "control" MVMs were needed. It can be seen from the computations with ATPRES that the linear system becomes more critical with increasing R: the number of MVMs and the computation time increase with R. For these computations the number of MVMs per LINSOL call was limited to 150 000 MVMs. None of the unpreconditioned methods PRES20, BICO, and BiCGSTAB converged for any value of R. For our `threshold2` tests we selected the value $R = 1000$.

Table 3 shows the tests with different values of `threshold2` for $R = 1000$. Here we have a behavior quite different from that in 2-D, although the number of unknowns and the width of the band are quite similar. Because of the low number of 13 grid lines in the y and z-directions the matrix A is more "compact" and consequently the Gauss factor GF and the fill-in factor FIF are smaller than in 2-D for full UL. However, if we increase `threshold2`, GF, FIF

Table 3
Tests with `threshold2` for (I)UL, $R = 1000$.

`threshold2`	GF	FIF	$\sharp(U+L)$ in Mwords	MVMs	total time in seconds	
0(UL)	23.7	63.9	16.4	4	2013	
$3 \cdot 10^{-8}$	23.7		16.5	4	2029	
10^{-7}	23.7		16.3	4	2120	
$3 \cdot 10^{-7}$	23.6		16.3	5	2115	
$6 \cdot 10^{-7}$	23.6		16.3	5	2108	
10^{-6}	23.6		16.3	5	2071	
$2 \cdot 10^{-6}$	23.5		16.2	8	2319	↓
$3 \cdot 10^{-6}$	23.5		16.2	12	2444	increase
10^{-5}	23.3		16.1	24	3157	in time, but
$1.5 \cdot 10^{-5}$	23.2		16.0	48	4932	no reduction
$1.7 \cdot 10^{-5}$	23.2	59.1	16.0	94	7841	in storage
$1.8 \cdot 10^{-5}$	23.2	58.8	16.0	110	9253	
$2 \cdot 10^{-5}$	23.1	58.6	16.0		$> 4\ h$	
				ATPRES	5875	robust, slow

and the storage for $U + L$ decrease very slowly while for `threshold2` $> 10^{-6}$ the number of iteration steps and thus the computation time increase rapidly. Because there is nearly no decrease in the storage for $U + L$, there is no minimum in computation time like in the 2-D examples. Thus full UL is optimal and IUL has no advantage, neither in time nor in storage. For `threshold2` $= 2 \cdot 10^{-5}$ the storage for $U + L$ has decreased from 16.4 Mwords to 16.0 Mwords; nevertheless the preconditioned iteration does no longer converge. Compare the value of $3 \cdot 10^{-5}$ in Table 1 of the 2-D case. Here the storage for $U + L$ had decreased from 29.2 Mwords to 9.7 Mwords and the computation time from 3168 sec to 2808 sec. So this 3-D case does not at all behave as we expected.

The exact maximum relative error for this example is $0.15 \cdot 10^{-1}$, but the error estimate shows more than 100% error. This means that the order 6 by which the error is estimated, is by far overdrawn on that coarse grid.

We also ran the example of full UL of Table 3 with the new improved code. The total CPU time reduced from 2013 sec to 1813 sec, which is 10% less time. So with the new code the timings of Table 3 would be reduced by roughly 10%.

6 Concluding remarks

The present investigations are work in progress. Remember our essential restriction that all conclusions are for our present implementation and essential improvements in different parts of the code may change our evaluation of (I)UL preconditioning. On the other side, improvements in the pure CG-type algorithms that create faster and more robust methods may avoid the necessity of the expensive (I)UL preconditioning at all. The essential drawback of (I)UL is the high price for the factorization $A = UL$ or $A = U^\star L^\star + R$ that is a fixed "setup" time, combined with large storage requirements for U and L or $U^\star$ and $L^\star$.

Certainly (I)UL pays only in very critical cases where all "fast" CG-type methods fail. We have selected a typical 2-D and 3-D case as they appear in the application of FDEM to CFD problems. In the 2-D case IUL was clearly superior to UL, see Table 1; and one has additionally the freedom to trade computation time for storage. However, we need at least a GF of 12 or a FIF of 15 which means an order of magnitude more storage than the original matrix! If we compare with ATPRES, this method requires the 1.6-fold computation time of the optimal IUL, but it needs no additional storage. So in this case ATPRES is the better choice if memory plays an essential role. In 3-D, see Table 3, we have no choice between UL and IUL because completely unexpected neither the GF nor the FIF reduces significantly with increasing value of `threshold2`, but soon the number of iterations goes up very fast and consequently also the computation time. The ATPRES method needs roughly the 3-fold computation time of full UL, but the latter has a GF of 23.7 or a FIF of 63.9. The large ratio between FIF and GF says that there were many computed zeros in $\tilde{A}$. So in the 3-D case one has the choice to pay the 3-fold computation time with ATPRES, but no additional storage. The smaller computation time of UL must be paid by the 24-fold storage of the original matrix. Thus for very large problems there is no alternative to ATPRES.

Before we did these experiments, when we saw the slow convergence and large computation time of ATPRES, we expected to cure the slow convergence essentially by (I)UL. However, as we now see the results with (I)UL, both in computation time and storage, we must change our opinion. So we think in the present situation that the results speak strongly in favor of ATPRES.

In the future we will surely improve the code, both for (I)UL and for ATPRES, but this will not change the situation basically. There is an urgent need for generalized CG methods that are as robust as ATPRES, but much faster. We will continue on our way to search for such methods.

References

[1] Willi Schönauer, Scientific Computing on Vector Computers, North-Holland (Amsterdam, 1987)

[2] Willi Schönauer, Hartmut Häfner, Rüdiger Weiss, LINSOL, a parallel iterative linear solver package of generalized CG-type for sparse matrices, in M.Heath et al. (Eds.), Proc. of the Eighth SIAM Conf. on Parallel Processing for Scientific Computing, SIAM (Philadelphia, 1997), CD-ROM (ISBN 0-89871-395-1), 8 pages.

[3] Hartmut Häfner, Willi Schönauer, Rüdiger Weiss, The parallel and portable linear solver package LINSOL, in H. Lederer, E. Hertweck (Eds.), Proc. of the Fourth European SGI/Cray MPP Workshop, IPP R/46, October 1998, Max-Plauck-Institut für Plasmaphysik (Garching bei Mñchen, Germany), pp. 242–251.

[4] Hartmut Häfner, Willi Schönauer, Rüdiger Weiss, The program package LINSOL - Basic concepts and realization, Applied Numerical Mathematics, vol. 30, no. 2–3, 1999, pp. 213–224.

[5] Rüdiger Weiss, Parameter-Free Iterative Linear Solvers, Akademie-Verlag (Berlin, 1996).

[6] Willi Schönauer, Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers, self-edition W. Schönauer, Karlsruhe, Germany, ISBN 3-00-005484-7, see also http://www.uni-karslruhe.de/~rz∅3/book/

[7] Lutz Grosz, Christian Roll, Willi Schönauer, A black-box solver for the solution of general nonlinear functional equations by mixed FEM, in M. Krizek, P. Neitaanmäki, R. Stenberg (Eds.), Finite element methods, Fifty Years of the Courant Element, M. Dekker (New York, 1994), pp. 225–234.

[8] Willi Schönauer, Torsten Adolph, How WE solve PDEs, to appear in J. Comp. Appl. Math. (2000).

[9] Willi Schönauer, Torsten Adolph, Development of a black-box solver for nonlinear systems of elliptic and parabolic PDEs, in P.Neittaanmäki, T.Tiihonen, P.Tarvainen (Eds.), Proc. 3rd European Conf. Numerical Mathematics and Advanced Applications, World Scientific Publishing (Singapore, 2000), pp. 698–706.

[10] Willi Schönauer, Torsten Adolph, The FDEM (finite difference element method) program package: Fully parallelized solution of elliptic and parabolic PDEs, Proc. of ECCOMAS 2000, Barcelona, CD-ROM ECCOMAS 2000, Laboratori de Calcul de la Facultat d'Informatica de Barcelona (FIB, UPC), ISBN 84-89925-70-4, 20 pages.

[11] Yousef Saad, Iterative Methods for Sparse Linear Systems, PWS Publishing Company (Boston, MA, 1996).

[12] Anne Greenbaum, Iterative Methods for Solving Linear Systems, SIAM (Philadelphia, PA, 1997).

[13] Jack Dongarra, Iain Duff, Danny Sorensen, Henk van der Vorst, Numerical Linear Algebra for High-Performance Computers, SIAM (Philadelphia, PA, 1998).

[14] Gerard Meurant, Computer Solution of Large Linear Systems, Elsevier (Amsterdam, The Netherlands, 1999).

[15] Matthias Bollhöfer, Yousef Saad, LUs and factorized approximate inverses are strongly related. Part I: Overview of results, University of Minnesota Supercomputing Institute Research Report UMSI 2000/39.

[16] Michele Benzi, Miroslav Tuma, A comparison study of sparse approximate inverse preconditioners, Report LA-UR-98-0024, Los Alamos National Laboratory (Los Alamos, NM, 1998).

[17] Hartmut Häfner, Willi Schönauer, The integration of different variants of the (I)UL algorithm in the LINSOL program package, in preparation.

[18] Henk van der Vorst, BiCGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of non-symmetric linear systems, SIAM J.Sci. Statist. Comput. 13, 1992, pp. 631–644.

[19] Willi Schönauer, Rüdiger Weiss, An engineering approach to generalized conjugate gradient methods and beyond, Applied Numerical Mathematics 19, 1995, 175–206.

[20] Louis Hageman, David Young, Applied Iterative Methods, Academic Press (New York, 1981).