# The Program Package LINSOL - Basic Concepts and Realization

Hartmut Häfner, Willi Schönauer, Rüdiger Weiss

*Numerikforschung für Supercomputer, Rechenzentrum Universität Karlsruhe, Postfach 6980, 76128 Karlsruhe, Germany, e-mail: haefner@rz.uni-karlsruhe.de*

**Abstract**

LINSOL is an iterative linear solver package (solving $Ax = b$) with direct solvers as preconditioners. The program package is adapted to the application of sparse matrices, but can be efficiently applied to full matrices, too. It contains presently fourteen iterative methods and nine polyalgorithms of generalized Conjugate Gradient (CG) methods. The used direct solver is the Gaussian algorithm working on the skyline of the matrix of the linear system.

Ten different data structures are supported by LINSOL to ease the embedding of LINSOL into an application as well as the mapping of arbitrary sparse matrices to storage patterns.

*Key words:* mathematical software, linear systems, iterative solvers, direct solvers.

## 1 Basic Concepts of LINSOL

Some fundamtental rules have been followed designing easy-to-use and efficient mathematical (super)computer software:

- **Flexibility or ease of use** - by the design of LINSOL that enables the user to use three types of interfaces and by the support of ten different data structures and storage patterns, respectively.
- **Portability** - by using the standardized Fortran77 and Fortran90 programming languages on single processor systems and additionally the *message passing* paradigm on parallel computers.
- **Robustness** - is obtained by fourteen different iterative solvers (excluding the polyalgorithms) based on generalized CG methods; they are suitable for different classes of matrices A; the Gaussian algorithm should be used as *emergency exit*, if all iterative methods fail.
- **Efficient code** for workstations (PCs), vectorcomputers and parallel computers - by reusing data in the highest memory hierarchy as far as possible and by

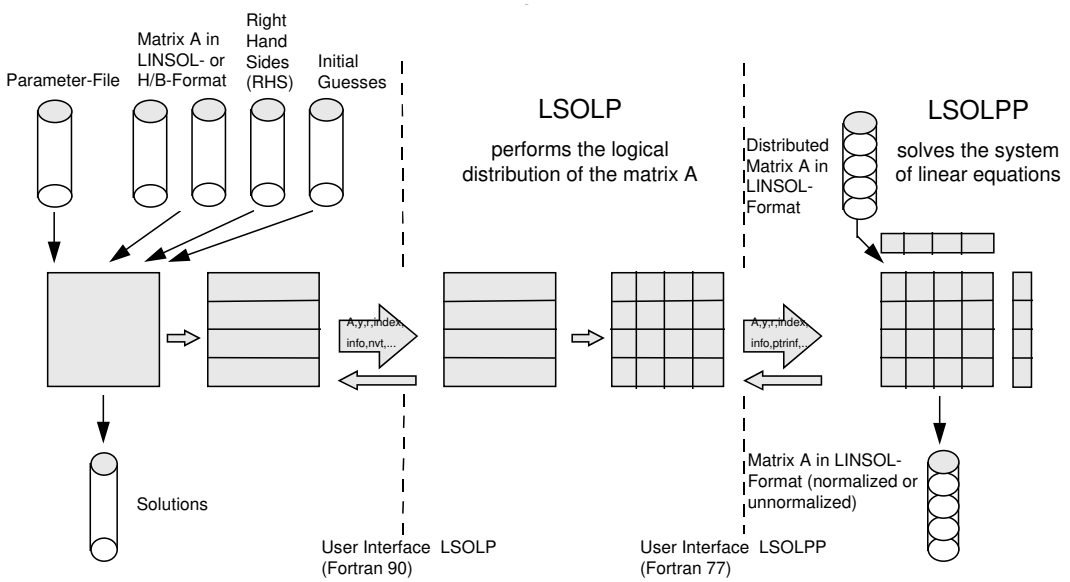


Fig. 1. The Design of LINSOL.

using vector pipelining and parallel algorithms with highest possible *volume-to-surface* ratio and designing them so that *latency hiding* is possible.

## 2 The Design of LINSOL

LINSOL is designed so that three types of interfaces can be used. Using the standard interface, parameters must be defined in a file controlling the program flow of LINSOL. The matrix $A$ can be stored in Harwell/Boeing- or LINSOL-format. The LINSOL-format has been developped with the aim of minimizing the storage requirements of a single processor on parallel computers. For the storing the matrix is read in partitions of vectors corresponding to the data structures of LINSOL; on parallel computers each vector, i.e. each basic data structure like diagonals, rows or columns, is divided into vector parts that belong to different processors; then these vector parts are sent immediately to the respective processors. As a consequence this method enables the reading of a matrix, to be distributed on many processors, on a single processor, if the main memory of one processor is too small to store the whole matrix.

An arbitrary number of *Right Hand Sides* must be stored in an additional file. The use of *Initial Guesses* is optional. On parallel systems the solution vectors are gathered on one processor. Their number corresponds to the number of *Right Hand Sides* and they are written into a new user-specified file.

The Fortran90 user interface LSOLP enables the user to integrate the program

package into an application. To run LSOLP on more than one processor the matrix $A$ and the vectors $x$ and $b$ must be distributed onto the processors. The Fortran77 user interface LSOLPP can be used on systems lacking a Fortran90 compiler.

All commonly used storage patterns for sparse matrices can be mapped to one of the ten data structures of LINSOL; these basic data structures are: main diagonal (assumed full), full (non-main) diagonal, packed diagonal (elements and their indices, from the first to the last nonzero element), indexed column (one element per row and the corresponding column index, from the first to the last nonzero element), indexed row (similarly one element per column and its row index), starry sky (nonzero elements with row and column indices), full row, full column, packed row (nonzero elements in a row and their column indices from the first to the last nonzero element) and packed column (similarly nonzero elements in a column and their row indices). They can be assembled without cutback to the matrix $A$, if $A$ is nonsymmetric. A symmetric matrix $A$ is stored in a special storage scheme using the symmetry of the matrix. Matrix $A$ can be represented as

$$A = A_s + D + A_s^T,$$

where $D$ is the main diagonal of $A$ and $A_s^T$ denotes the transpose of $A_s$. If $A$ is symmetric, only the matrix $A_s$ and the main diagonal $D$ must be handed over to LINSOL. Assembling $A_s$ all basic data structures can be used with the cutback that main diagonal elements must be stored in $D$ - equivalent to the basic data structure "main diagonal". Thus the user interfaces LSOLP and LSOLPP can be embedded easily (in less than one day) into an application program. The storage patterns of LINSOL except "main diagonal" are shown in Fig. 2.

## 3 Portability

On single processor systems portability is achieved by using the standardized programming languages Fortran77 and Fortran90.

At present the only possible choice is the *message passing* paradigm to implement portable parallel programs. All manufacturers of MIMD computers offer some *message passing* libraries, but the problem is that these libraries have no common subset of basic *message passing* routines. There are two ways to solve this problem.

The first one is to use PVM or MPI. On nearly all MIMD computers PVM and MPI are available, so that this is now the most comfortable way to implement portable parallel programs. For performance reasons MPI should be used.

The second one - which has been chosen by us - is to implement an own *Portable Message Passing Interface* (P_MPI) [3]. We only defined interfaces for a few im-

Basic data structures of the matrix A supported by LINSOL and the corresponding basic vector operation types of the matrix-vector multiplication (c=A*r):

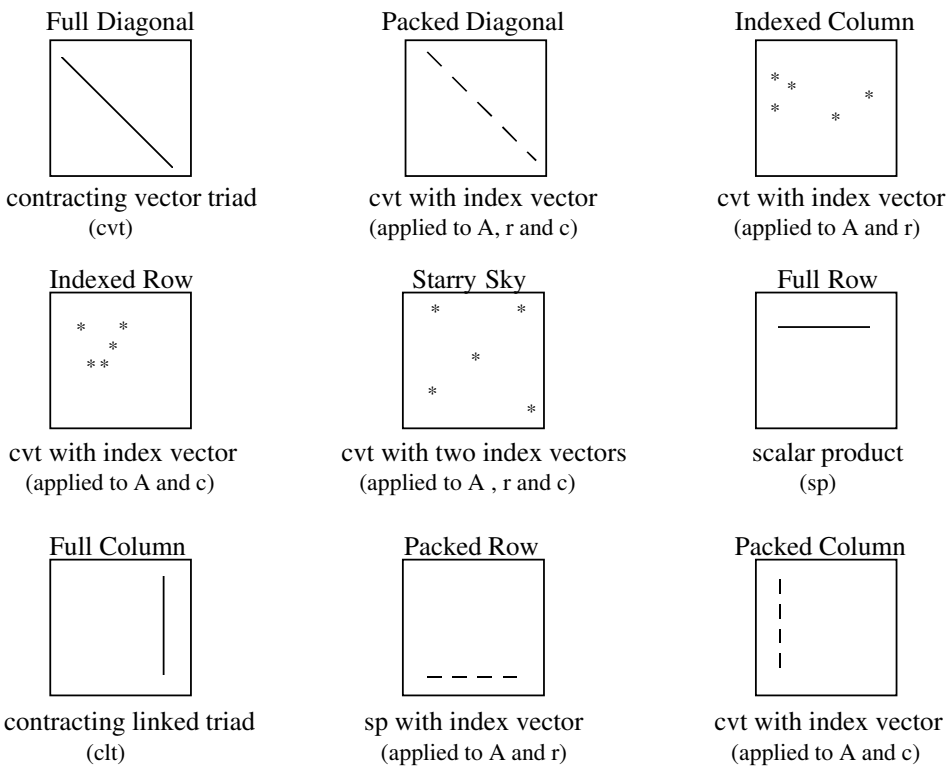


Fig. 2. Basic Data Structures of LINSOL.

portant basic *message passing* routines with parameters being common to all well-known *message passing* interfaces. Thus we only have to write down our calls for the **send** and **receive** routines of P_MPI and to set up a few communication parameters like e.g. a table for the logical processor numbers and the according physical processor numbers within the start-of-communication routine to obtain a portable code. Since we provide interfaces to all well-known *message passing* interfaces the user only has to change the name of the *message passing* interface in the link step to get another *message passing* interface. So the user can get the highest possible performance rates (concerning the communication network) for all MIMD computers.

## 4 Robustness

To get robustness fourteen different iterative solvers [7] enabling the solution of linear systems for most classes of matrices are implemented. Additionally, adaptive polyalgorithms are provided that choose an appropriate solver from the fourteen methods. The used iterative methods are Krylov subspace methods. There are five

different classes of methods:

(1) methods for symmetric and positive definite matrices: classical CG and ORTHOMIN. The methods require one matrix-vector multiplication per iteration step.
(2) truncated and restarted methods for moderately nonsymmetric systems like GMRES, ORTHORES and PRES20. These methods require one matrix-vector multiplication per iteration step. PRES20 is a special version of smoothed ORTHORES and the iteration process is restarted every 20-th step. For sufficient diagonal dominance PRES20 is a quickly converging method.
(3) biconjugate gradient-based methods for nonsymmetric but definite systems like BCG, BICO, QMR, CGS and BiCGSTAB. All above mentioned algorithms have the same origin: BCG. All of them have the common property that two matrix-vector multiplication have to be performed in each iteration step and that the exact methods can be formulated with a short recurrence, i.e. the methods do not have to be restarted during the iteration. The latter property is one of the reasons for the popularity of BCG-based methods. BCG can be considered as exact CG method applied to a double system. The residuals and errors of BCG oscillate heavily. In order to obtain a better behaviour Schönauer applied minimal residual smoothing to BCG [4]. Then the algorithm is called BICO. QMR is implemented without the look-ahead process that is proposed in the original paper by Freund and Nachtigal [1]. The difference between QMR and BICO is the different kind of smoothing to be applied to BCG. All these methods including CGS and BiCGSTAB are closely related, because they have the same origin. Numerical tests confirm a similar behaviour of these methods. However, BiCGSTAB seems to be superior in many cases.
(4) methods based on the normal equations as emergency exit like CGNE and CGNR-ATPRES. Both methods require two matrix-vector multiplications, one by $A$ and one by $A^T$. For these two methods the condition number is squared, i.e. the convergence is slow for matrices close to symmetric matrices. Theoretically both methods converge always. Thus they are very robust, but not very efficient.
(5) error-minimizing methods like GMERR. Three variations of this algorithm are implemented in LINSOL: GMERR(5), GMERR(20), GMERR(S). GMERR(S) is Fridmann's method and can only be applied to symmetric matrices. GMERR(5) means that the GMERR method is restarted every fifth step; GMERR(20) restarts GMERR every 20-th step. Both restarted versions can be applied to nonsymmetric matrices. To avoid the oscillation of the residuals, residual-minimizing smoothing is applied.

Smoothing leads to a monotonical decrease of the residuals and thus can improve the qualitative convergence, even if it cannot improve the quantitative convergence, i.e. the convergence is not essentially accelerated by smoothing; however, a better control of the iteration process results.

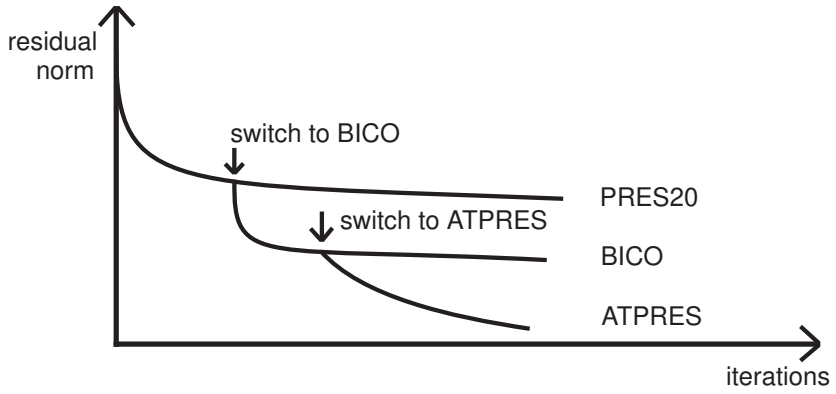


Fig. 3. Polyalgorithm I.

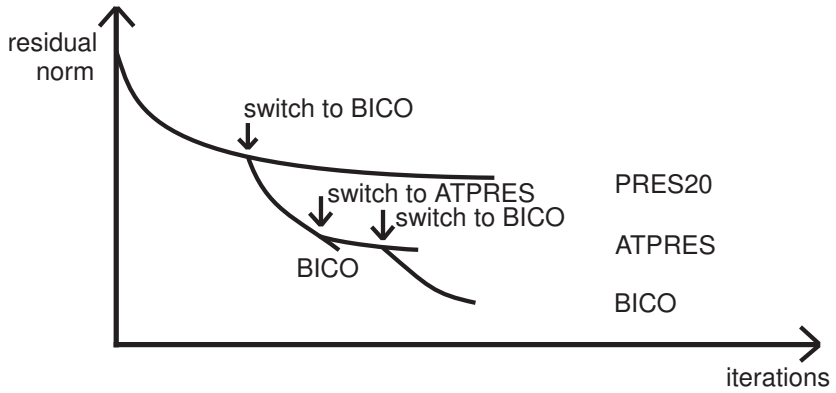


Fig. 4. Polyalgorithm II.

All these methods are explained with their properties in detail in [6].

The fourteenth algorithm solves the linear system $A^T Ax = b$ with the matrix $A$ as input matrix. LINSOL internally exploits the symmetry of the matrix $A^T A$.

Furthermore, LINSOL supports different kinds of polyalgorithms. A key for the development of these techniques is the perception that the full set of iterative methods can be represented by a quite smaller set. It seems to be reasonable to select one representative method of the truncated or restarted CG methods (class 1), one representative of BCG-based methods (class 2) and one of methods based on normal equations (class 3); the polyalgorithms in LINSOL monitor the norms of the residuals and switch from efficient but less robust methods to more robust but less efficient methods. Polyalgorithm I and II start with PRES20 (see Fig. 3 and 4). If the reduction of the residual is not sufficient after certain steps, then BICO is

applied. If BICO converges too slowly polyalgorithm I switches to the emergency exit ATPRES, whilst polyalgorithm II checks the convergence behaviour and returns to PRES20, if its convergence behaviour was better than that of BICO, else it switches to ATPRES, too. Polyalgorithm I never changes from ATPRES to another algorithm; polyalgorithm II again checks the convergence behaviour of ATPRES against that of BICO and returns to BICO, if it is better. Three more polyalgorithms call GMERR(5) or GMERR(20) before calling ATPRES and are implemented without or with checking of the convergenge behaviour. Polyalgorithm VI (VII) starts with GMERR(5) (GMERR(20)) and switches to ATPRES; polyalgorithm VIII additionally checks the convergence behaviour. Polyalgorithm IX starts with GMERR(S) and switches to the *classical* CG method; it is only suited to symmetric matrices. Over all nine polyalgorithms are implemented in LINSOL. For all methods and polyalgorithms hold, that no information about the eigenvalues or other inner properties of the matrix is required.

If all iterative methods fail, the full Gaussian algorithm implemented as preconditioner can be used as *emergengy exit*. Future versions of LINSOL will also contain ILU-algorithms (ILU stands for Incomplete LU). Now a brief mathematical outline follows how the Gaussian algorithm is embedded into the iterative algorithms.

Instead of the linear system $Ax = b$ we have to solve the preconditioned system $P_L A P_R y = P_L b$ with $x = P_R y$. $P_L$ $(P_R) \in R^{n \times n}$ is a non-singular left-hand (right-hand) preconditioning matrix acting on the equations (solutions). By the Gaussian elimination we get the triangular matrices $U$ and $L$ with $A = U * L$. With $P_L = U^{-1}$ and $P_R = L^{-1}$ we get the optimal choice $P_R P_L = L^{-1} U^{-1} = (UL)^{-1} = A^{-1}$. Using $P_L$ in the iteration process, i.e. to get a preconditioned vector $r^P$ from the original vector $r$, means to preform a forward elimination step; using $P_R$ in the iteration process, i.e. to get the solution $x$ from the solution $y$ of the preconditioned system, means to perform a backward substitution step:

$$\begin{aligned} Ur^P = r \Longleftrightarrow U^{-1} r = r^P \Longleftrightarrow P_L r = r^P \ , \\ Lx = y \Longleftrightarrow L^{-1} y = x \Longleftrightarrow P_R y = x. \end{aligned}$$

As example preconditioning is described in connection with the *classical* CG algorithm for a symmetric positive matrix $A$. In comparison to the *classical* CG algorithm the residuum $r_k (k = 0, 1, 2, ...)$ has to be substituted by $r_k^P = P_L r_k$ and the matrix $A$ by $P_L A P_R$. Thus the term $Ar_k$ must be substituted by $\hat{r}_k = P_L A P_R r_k^P$.

For an arbitrary *initial guess* $x_0$ holds:

$$Ax = A(\tilde{x} + x_0) = b \Longleftrightarrow A\tilde{x} = b - Ax_0 \Longleftrightarrow A\tilde{x} = \tilde{b}.$$

Now we solve $P_L A P_R \tilde{y} = P_L \tilde{b}$ with $\tilde{x} = P_R \tilde{y}, \tilde{x}_0 = 0$ and calculate

$$
\begin{aligned}
&r_0 = -\tilde{b},\ r_0^P = P_L r_0,\ \hat{r}_0 = P_L A P_R r_0^P \text{ and for } k \geq 0 : \\
&\alpha_{1,k}^P = -\frac{\left(r_k^P\right)^T \hat{r}_k}{\left(r_k^P\right)^T r_k^P} \quad , \\
&\alpha_{2,k}^P = -\frac{\left(r_{k-1}^P\right)^T \hat{r}_k}{\left(r_{k-1}^P\right)^T r_{k-1}^P} \quad , \\
&\phi_k^P = \frac{1}{\alpha_{1,k}^P + \alpha_{2,k}^P} \quad , \\
&r_{k+1}^P = \phi_k^P (P_L A P_R r_k^P + \alpha_{1,k}^P r_k^P + \alpha_{2,k}^P r_{k-1}^P) \\
&\qquad\qquad \text{with } r_k^P = P_L r_k \ , r_k^{PP} = P_R r_k^P \text{ and } \hat{r}_k = P_L A P_R r_k^P \quad , \\
&x_{k+1} = P_R y_{k+1} = \phi_k^P (P_R r_k^P + \alpha_{1,k}^P P_R y_k + \alpha_{2,k}^P P_R y_{k-1}) \\
&\qquad\qquad = \phi_k^P (r_k^{PP} + \alpha_{1,k}^P x_k + \alpha_{2,k}^P x_{k-1}).
\end{aligned}
$$

As the whole iteration process runs in the - by the matrix $P_L$ - shifted space, the only additional computational cost is one forward elimination step with $P_L$ and one backward substitution step with $P_R$ per iteration step.

The Gaussian elimination process only operates on the skyline of the matrix A to accelerate the computation and minimize the storage requirements. To get a *small* skyline of the matrix $A$ a bandwidth optimizer is used in a preprocessing step.

## 5 Optimization Strategies

The following principles [4] have been considered for the design of LINSOL:

- Keep the floating-point units continuously busy. As this is a mostly unattainable goal, this principle can be reformulated into a practical rule: Reuse data in the highest memory hierarchy as far as possible. This rule has been realized in LINSOL by the implementation of cache reuse strategies for the matrix-vector multiplication.
- Use the principle of the separation of the selection and processing of the data. Thus processing can take place with vectors of contiguous or (and) indexed elements. This rule has been realized implicitly in LINSOL by supporting data structures for all commonly used storage patterns for sparse matrices.

The consequent use of vector operations in LINSOL leads not only to an optimal code for vectorcomputers, but also for workstations with superscalar processors; for cache-based systems like workstations there is a switch in LINSOL to enhance cache reuse.

These single processor optimizations are a prerequisite for an optimal parallel code.

LINSOL has been developped as "black box" software. Thus the parallelization of the used methods must be transparent to the users. This goal has been achieved by the automatic distribution of the matrix $A$ on a single processor onto matrix blocks on many processors. Changing the number of processors means to change the data the processor works on; it does not mean to change the algorithm. Thus the convergence rate is the same - apart from rounding errors - for all numbers of processors. Adapted to the distribution of the matrix $A$ all vectors of LINSOL are distributed in vector parts onto the processors. This leads to **scalability regarding the memory**, which is an absolutely essential feature for parallel software.

All iterative methods implemented in LINSOL use as basic operations

- matrix-vector multiplications,
- reduce operations like dot products and
- vector operations.

The parallel key operation is the parallel matrix-vector multiplication. The overall goal for the implementation of this operation is to achieve scalability regarding the computation time by minimizing communication, skipping unnecessary communication cycles and hiding the communication behind the calculation on parallel computers with asynchronous communication (latency hiding).

To minimize communication, i.e. to hold the matrix $A$ locally on the single processors, the matrix $A$ is distributed in $p$ row blocks of nearly the same size onto the $p$ processors. Additionally each row block of the matrix $A$ is subdivided into $p$ logical column blocks corresponding to the distribution in row blocks, Fig. 5.

Thus the matrix-vector multiplication $c = A * r$ is computed concurrently on the $p$ physically distributed row blocks - all processors working on different column blocks. The mathematical formulation of the parallel matrix-vector multiplication is:

$$c^i = c^i + \sum_{j=1}^{p} A^{i,j} \cdot r^j.$$

In Fig. 5 the matrix-vector multiplication is depicted on 4 processors. Each processor $i(i = 1, .., p)$ must multiply sequentially different column blocks $A^{i,j}(j = 1, .., p)$ with the corresponding vector parts $r^j$. Between these sequential computation steps the vector parts $r^j$ must be sent around in a ring shift. Diagonal matrix blocks (e.g. $A^{1,3}$, $A^{2,4}$, $A^{3,1}$, $A^{4,2}$) are computed concurrently, if at least one of these blocks contains non-zero matrix elements. If all diagonal matrix blocks contain only zero matrix elements (e.g. in Fig. 5 $A^{1,2}$, $A^{2,3}$, $A^{3,4}$, $A^{4,1}$), the computation of all these diagonal matrix blocks can be skipped, i.e. the number of communica-

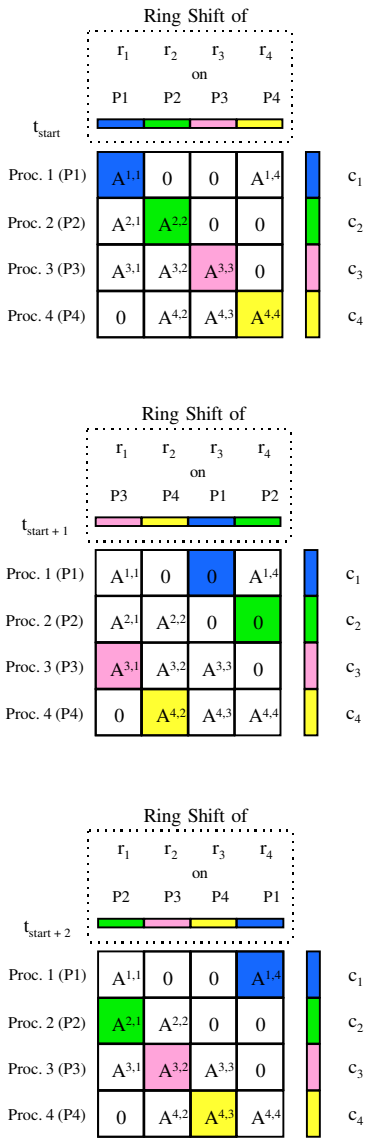


Fig. 5. The Parallel Matrix-vector Multiplication on 4 processors. Active blocks are shaded.

tion cycles is reduced by one. Double buffer technique is used to enable overlapped processing of computation and communication on parallel computers with independent communication units.

For many algorithms like BCG, BICO, QMR, CGNE and CGNR-ATPRES the matrix-vector multiplication by the transpose of $A$ ($d = A^T r$) is additionally required. Since the matrix $A$ is not transposed physically, now the parts of the result vector have to be sent around in a ring shift. Whilst the contribution of a block is computed, the part of the result vector is transferred, then the contribution is added and the new partial result is sent again. The parts of the input vectors now don't have to be moved around. Thus on each processor $i (i = 1, .., p)$ different vector

parts of $d$ are computed by

$$d^j = d^j + \sum_{j=1}^{p} A^{i,j^T} \cdot z^i = d^j + \sum_{j=1}^{p} A^{j,i} \cdot z^i.$$

Again the double buffer technique is used.

This strategy of parallelization of the matrix-vector multiplication leads to a high *volume-to-surface* ratio depending on the sparsity of the matrix $A$ and thus the software gets **scalable regarding the computation time**.

As all vectors are distributed onto the $p$ processors, scalar products are computed via a cascade over all processors and immediately broadcasted back to all processors. In order to reduce the communication overhead independent reduce operations can be calculated locally and then all independent partial and final results are sent by one and only one message on each processor involved in the cascadic operation.

Vector operations can be executed independently for the corresponding parts of the vectors that are distributed on different processors.

## 6 Measurements on an IBM SP and CRAY T3E-900

As test example we take matrices that are generated by the FEM program package VECFEM [2] applied to the system of 3-D partial differential equations of computational mechanics for the unit cube under surface load. A 3-D grid is generated so that the number of unknowns per processor approximately stays constant for all processor numbers. As the resulting matrix is symmetric, only half the non-zero elements except of the main diagonal are stored and the *classical* CG algorithm is used in LINSOL. The characteristics of the test cases are compiled in Table 1 (*e* means global no. of elements, *n* means global no. of unknowns, *m* means no. of non-zero elements in matrix, *op* means no. of operations, *it* means no. of iterations in LINSOL).

For test case $i, i = 2, 4, 8, 16, 32$ we get approximately the $i$-fold number of unknowns in comparison to the test case 1. For this type of problem the bandwidth, relative sparsity and the condition number of the matrix increases with the problem size. With the condition number the number of iterations in LINSOL increases. Increasing $n$ means to get more critical problems.

In Table 2 and 3 we present only two elapsed timings in seconds: $tc : p$ means that test case $tc$ runs on $p$ processors, $t_{CG}$ is the total time for CG and $t_{it}$ is the time for one CG iteration. In addition to the MFlop/s rate the MFlop/s ratio to the single processor test case is presented. In Fig. 6 the ratio of MFlop/s for the T3E-900, based on the last column of Table 3, is depicted.

| *test cases* | *e* | *n* | *m* | *op* | *it* |
|---|---|---|---|---|---|
| 1 | 1872 | 26793 | $4.14 * 10^6$ | $8.98 * 10^6$ | 513 |
| 2 | 3825 | 52800 | $8.38 * 10^6$ | $18.13 * 10^6$ | 657 |
| 4 | 8000 | 107163 | $17.38 * 10^6$ | $37.55 * 10^6$ | 778 |
| 8 | 15625 | 204828 | $33.73 * 10^6$ | $72.79 * 10^6$ | 974 |
| 16 | 31713 | 408192 | $68.12 * 10^6$ | $146.85 * 10^6$ | 1323 |
| 32 | 64000 | 811923 | $136.93 * 10^6$ | $294.97 * 10^6$ | 1561 |

Table 1
Characteristics of the test cases.

| $tc : p$ | $t_{CG}$ | $t_{it}$ | *MFlop/s* | $ratio_{MFlop/s}$ | $t_{CG}$ | $t_{it}$ | *MFlop/s* | $ratio_{Mflop/s}$ |
|---|---|---|---|---|---|---|---|---|
| 1:1 | 130.9 | .255 | 35.2 | 1 | 104.8 | .204 | 44.0 | 1 |
| 2:2 | 184.2 | .280 | 64.8 | 1.84 | 155.7 | .237 | 76.5 | 1.74 |
| 4:4 | 226.7 | .291 | 129.0 | 3.66 | 211.6 | .272 | 138.1 | 3.14 |
| 8:8 | 289.2 | .297 | 245.1 | 6.96 | 280.9 | .288 | 252.7 | 5.74 |
| 16:16 | 393.2 | .297 | 494.4 | 14.0 | 406.0 | .307 | 478.3 | 10.89 |
| 32:32 | 510.9 | .327 | 902.1 | 25.6 | 615.0 | .394 | 748.7 | 17.01 |

Table 2
Measurements on the IBM SP for $p$ = 1 to 32 processors (Power 2 wide nodes, 77 MHz, 308 MFlop/s peak performance, 256 KB Cache, 256 MB main memory). Left results without, right ones with cache reuse.

In [5] measurements on further supercomputers like Cray T90, NEC SX-4 and Fujitsu VPP300 are presented.

For this test example on the IBM SP cache reuse pays for small $p$, but does not pay for larger $p$. The reason is that the sparsity of the matrix $A$ increases the larger the problem is. Then the *cutting* in cache stripes costs more than it pays. It should be stressed that the Mflop/s ratio is not a speedup. To get the *classical* speedup we would have to solve the same problem for all measurements. But such a test example is not suited as the larger matrices $A$ do not fit into the memory of a single processor. The point is that the example shows the scalability of the program package LINSOL regarding the computation time.

Because of the small memory of 128 MB on the Cray T3E-900 in Jülich we had to use more processors for larger test cases than for the IBM SP. Without cache reuse we get a MFlop/s ratio which is greater than the theoretical speedup. This is due to a better cache reuse for matrices $A$ with increasing number of unknowns $n$. Switching on the implemented software module for cache reuse pays nearly always, Fig. 6. Additionally the performance increase is higher (till 50 percent) for small processor numbers than on the IBM SP.

| $tc : p$ | $t_{CG}$ | $t_{it}$ | *Mflop/s* | $ratio_{Mflop/s}$ | $t_{CG}$ | $t_{it}$ | *Mflop/s* | $ratio_{Mflop/s}$ |
|---|---|---|---|---|---|---|---|---|
| 1:1 | 229.3 | .447 | 20.1 | 1 | 145.6 | .284 | 31.6 | 1 |
| 2:2 | 287.2 | .437 | 41.5 | 2.06 | 208.8 | .318 | 57.0 | 1.80 |
| 2:4 | 130.1 | .198 | 91.6 | 4.56 | 104.9 | .160 | 113.3 | 3.59 |
| 4:8 | 149.5 | .192 | 195.6 | 9.73 | 137.8 | .177 | 212.1 | 6.71 |
| 8:16 | 177.0 | .182 | 400.0 | 19.90 | 166.0 | .170 | 428.2 | 13.55 |
| 8:32 | 89.4 | .092 | 791.2 | 39.36 | 88.1 | .090 | 808.8 | 25.59 |
| 16:64 | 129.0 | .098 | 1498.5 | 74.55 | 155.0 | .117 | 1255.1 | 39.72 |
| 32:128 | 164.4 | .105 | 2809.2 | 139.76 | 165.3 | .106 | 2782.7 | 88.06 |

Table 3
Measurements on the Cray T3E-900 in Jülich for $p$ = 1 to 128 processors (DEC Alpha EV-5, 450 MHz, 900 MFlop/s peak performance, 8 KB primary data cache and 96 KB secondary cache, 128 MB main memory). Left results without, right ones with cache reuse.

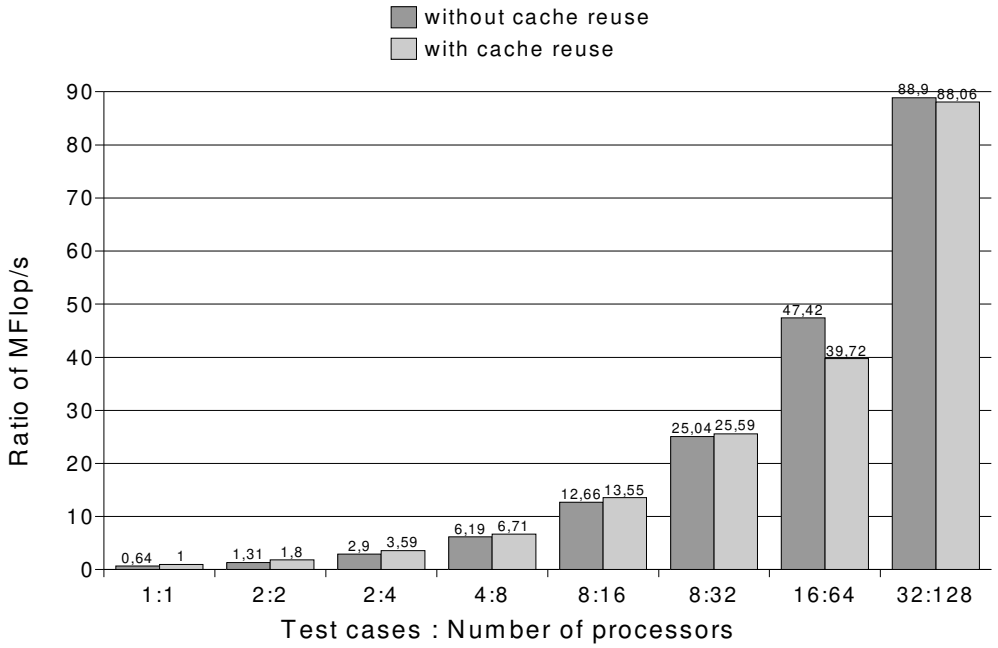


Fig. 6. The ratio of MFlop/s for the Cray T3E-900 based on the measurement of test case 1 on 1 processor with switched on cache reuse.

## 7 Concluding Remarks

LINSOL is a **black-box** iterative linear solver package. This means that firstly all used iterative solvers are parameter-free and secondly the users do not have to know anything about the internal data structures to use the LINSOL-library. The software is optimized for vectorcomputers and parallel computers, but it also runs efficiently on all UNIX- or LINUX-based workstations. The measurements on two well-known parallel computers show that LINSOL can solve extremely large problems on distributed memory parallel computers with an efficiency comparable to that of vectorcomputers.

Acknowledgement: We thank the HLRZ Jülich for the possibility to use their Cray T3E-900.

The references [3] [5] [7] are available in WWW under URL http://www.uni-karlsruhe.de/~LINSOL.

## References

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

[2] L. Grosz, C. Roll, and W. Schönauer. Nonlinear finite element problems on parallel computers. In J. Dongarra and J. Waśniewski, editors, *Parallel scientific Computing, First Int. Workshop, PARA'94*, pages 247–261, Denmark, 1994.

[3] H. Häfner. P_MPI User's Guide v1.0. Technical report, University of Karlsruhe, Computing Center, Postfach 6980, 76128 Karlsruhe, Germany, 1997.

[4] W. Schönauer. Scientific supercomputing. [Online.] Version 1.0. [Karlsruhe, Germany], 1998. HTML with running text as inline images in GIF format. Available from World Wide Web: <http://www.uni-karlsruhe.de/Uni/RZ/Personen/rz03/book/>.

[5] W. Schönauer, H. Häfner, and R. Weiss. LINSOL, a parallel iterative linear solver package of generalized CG-type for sparse matrices. In M. Heath et al., editors, *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia, USA, 1997. CD-ROM (ISBN 0-89871-395-1), 8 pages.

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

[7] R. Weiss, H. Häfner, and W. Schönauer. LINSOL (LINear SOLver)—description and user's guide for the parallelized version. Internal report 61/95, University of Karlsruhe, Computing Center, Postfach 6980, 76128 Karlsruhe, Germany, 1995.