The Parallel and Portable Linear Solver Package LINSOL

Hartmut Häfner, Willi Schönauer, Rüdiger Weiss

August 26, 1998

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 $Ax = b$). An integrated LU/ILU algorithm working as preconditioner is being implemented. 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 ten polyalgorithms of generalized Conjugate Gradient (CG) methods. Currently eight different normalization methods are used as preconditioner.

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.

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 based on generalized CG methods; they are suitable for different classes of matrices A; additionally, adaptive polyalgorithms are provided that choose an appropriate solver from the fourteen methods. To enhance the robustness of LINSOL an (incomplete) LU algorithm with an integrated bandwidth optimizer embedded as preconditioner into the iterative methods is being integrated into the program package.
- **Efficient code** for workstations (PCs), vectorcomputers and parallel computers - by reusing data in the highest memory hierarchy as far as possible and by 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. On p processors the matrix A can be stored distributed in p files. The matrix A stored distributed in p files can be read again on p processors to be computed locally. 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.

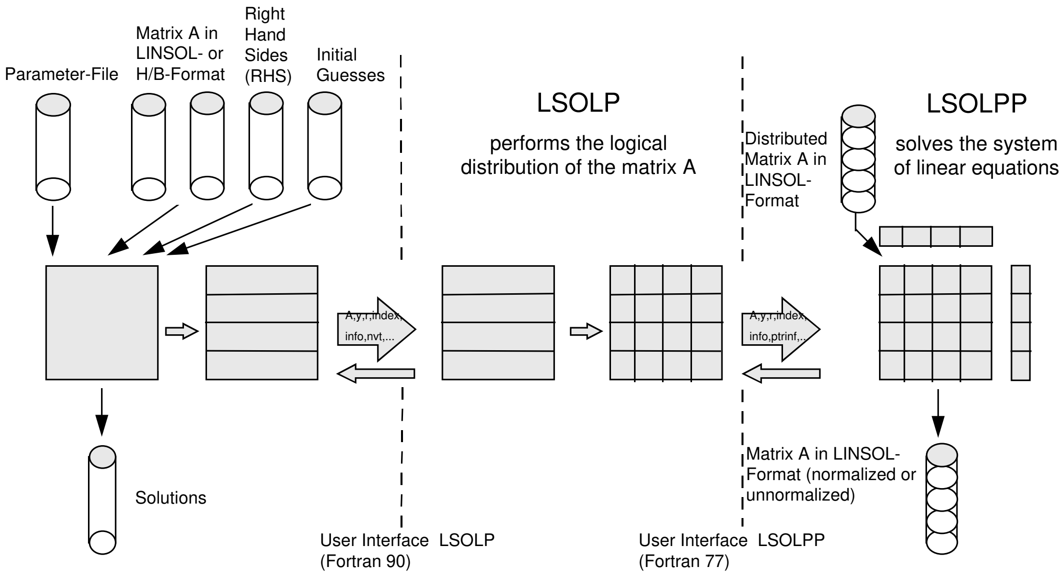

Figure 1: The design of LINSOL.

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) [4]. We only defined interfaces for a few important basic *message passing* routines like **send**, **receive** and communication-setup routines being common to all well-known *message passing* interfaces; the communication-setup routine sets a few communication parameters like e.g. a table for the logical processor numbers and the

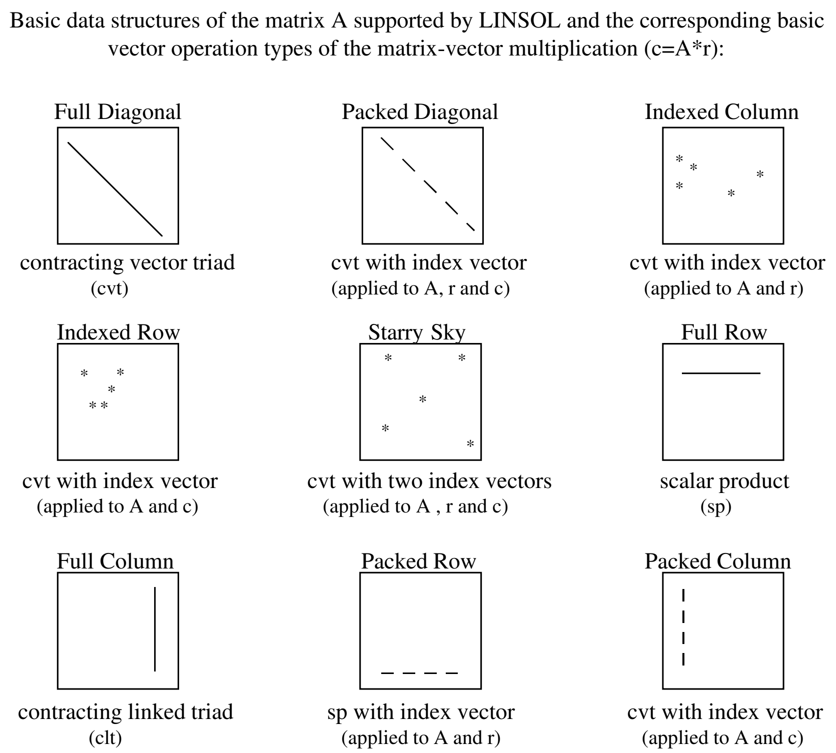

Figure 2: Basic data structures of LINSOL.

according physical processor numbers. 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.

Thus we have implemented a simple, synchronous *message passing* interface for the Cray T3E basing on the *shared memory put* operation. On each processor $2 * p$ buffers, 2 buffers for each of p processors, of a predefined length are initialized within this interface. The length determines the number of outstanding sends and receives respectively. For each of these $2 * p$ buffers exists a pointer to the next free location in the buffer scrolling through the buffer till its end to start again at the first buffer entry.

How does the *message passing* now work? We want ot send data from processor j to processor i. The **receive** routine called on processor i puts ($\equiv$ *shared memory put*) the three variables $mid, loc, msgtyp$ into the i-th send buffer on processor j from whom the matching send routine is started. The pointer to the i-th send buffer on processor j shows the next free location in the send buffer. The variable loc contains the memory address of the receiving processor where the data to be sent should be stored in. Then the **receive** routine sets the pointer to the next free location and returns.

The **send** routine scans through the i-th buffer until the variable $msgtyp$ identifying matching communication operations is found. Then it puts the data to be sent - start address and length of the data are parameters of the interface - into the memory of processor i starting with address loc. After that it puts the variable mid into the j-th receive buffer on the processor i. The pointer to the j-th receive buffer shows the next free location in the this buffer. Then the **send** routine clears the entry in the send buffer for the sent data, sets the pointer to the next free location and returns.

The **receive-wait** routine scans through the j-th receive buffer until the variable mid identifying matching receive and receive-wait routines is found. Then it knows that the sent data have arrived and returns.

Analyzing the protocol one can see that this *message passing* interface is **synchronous**. The **send** routine is **blocking** and can not complete before the matching receive has sent necessary information data. The **receive** routine is **nonblocking**. Fig. 3 shows exemplarily the *message passing* protocol on 3 processes.

A synchronous interface means that there are restraints one has to pay attention to. If e.g. data are sent in a ring shift, the alternate buffer technique must be used. Omitting this feature means to run into a deadlock. As LINSOL always uses this technique, synchronous *message passing* protocols can be employed; beyond it this feature allows

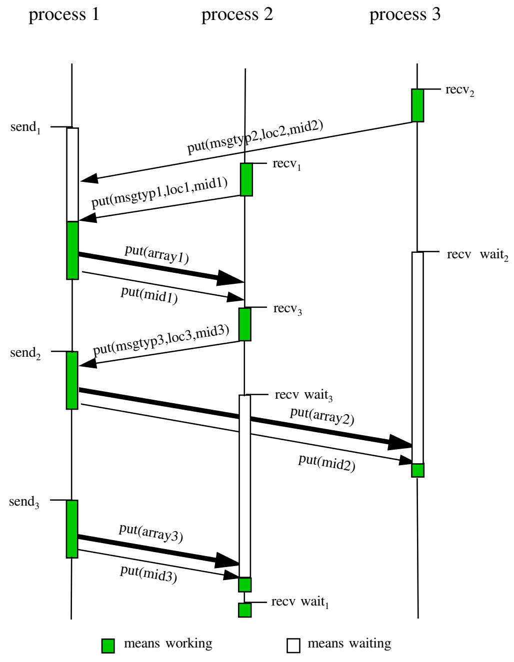

Figure 3: The *message passing* protocol on 3 processes.

the hiding of the computation behind the communication (latency hiding).

4 Robustness

To get robustness fourteen different iterative solvers [8] 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 [5]. 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.

All these methods are explained with their properties in detail in [7].

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. 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. The polyalgorithms IX and X start with GMERR(S) and switch to the *classical* CG method; they are only suited to symmetric matrices. Over all ten 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.

Currently an LU/ILU algorithm is being implemented as preconditioner (ILU stands for Incomplete LU). The Gaussian elimination process will only operate 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 will be used in a preprocessing step.

5 Optimization Strategies

The following principles [5] 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 by 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).

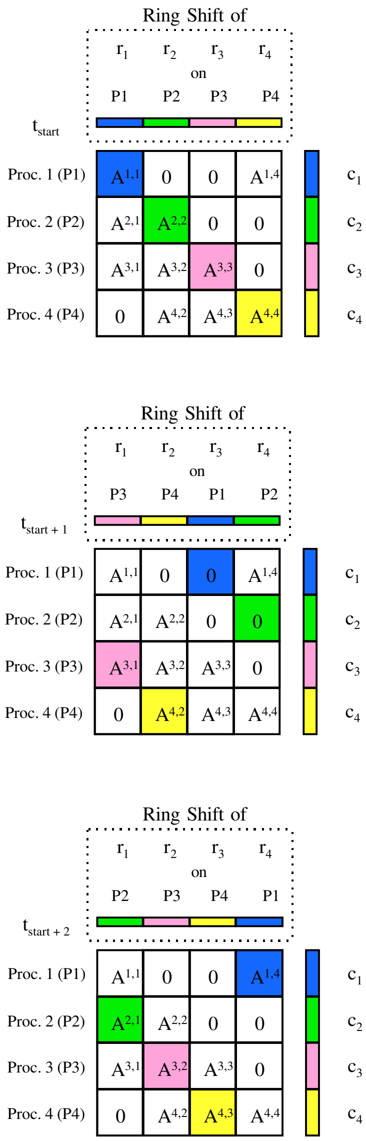

Figure 4: The Parallel Matrix-vector Multiplication on 4 processors. Active blocks are shaded.

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. 4.

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. 4 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. During 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 communication 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**.

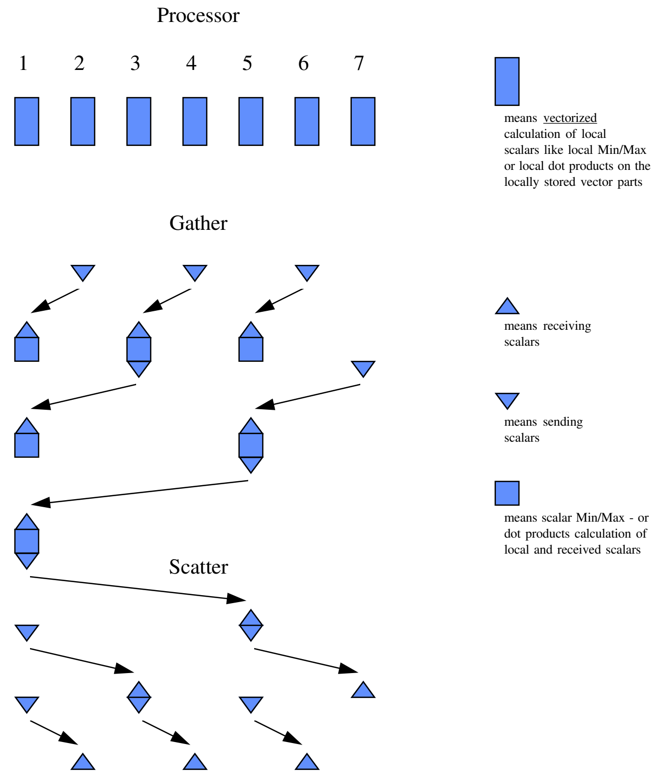

Figure 5: Computation of dot products on 7 processors.

While reduce operations like dot products are processed very efficiently on vectorcomputers, they become a critical operation on parallel computers. As all vectors are distributed onto the p processors, each processor first computes its local dot product(s); then a reduce routine is called that gathers scalars via a cascade over all processors; these scalars are newly computed as sums of the local and received scalars within the reduce routine. After this step the global dot product(s) are only available on the first processor. In the second part of the reduce routine this scalars are scattered onto all processors using the reverse cascade as for the gather operation (see Fig. 5).

In an optimized reduce routine the gather and scatter operations are performed concurrently on all processors, i.e. all processors - not only the first processor - compute the global dot products in a cascade (see Fig. 6). This optimization leads to a reduction of reduce steps by a factor of 2.

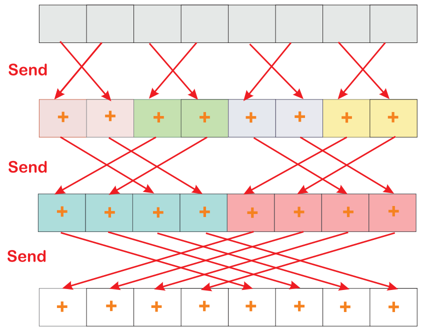

Figure 6: Optimized computation of dot products on 8 processors.

In order to reduce the communication overhead independent reduce operations can be executed 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 CRAY T3E-900

First we measured the simple and double *Ping-Pong* communication benchmark with standard nonblocking MPI-routines. For the simple (double) *Ping-Pong* we got a startup time of 23 (31) microseconds and a peak performance of 166 (147) MB/s. Then we performed the same measurements with the described synchronous *message passing* protocol basing on *shared memory put* routines. Now we got for the simple (double) *Ping-Pong* a startup time of 7 (11) microseconds and a peak performance of 300 (218) MB/s. Thus the communication time is reduced by a factor of 3 for very short messages and by a factor of 1.5 till 2 for very long messages.

Second we measured the computation of a dot product on 8 processors for different vectorlengths and compared the performance rates by linking the two different reduce routines. Using MPI we got a performance enhancement for the optimized reduce routines of factor 1.3 (5) for very long (short) vectors. Using the described synchronous *message passing* protocol we got a performance enhancement for the optimized reduce routines of factor 1.009 (1.3) for very long (short) vectors. The measurements show that the optimized reduce routine accelerates the computation of reduce operations on parallel computers.

We tried to enhance the performance of our test example with both optimization methods.

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.

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 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. 7 the ratio of MFlop/s for the T3E-900, based on the last column of Table 2, is depicted.

$tc : p$	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 2: Measurements on the Cray T3E-900 in Jülich with MPI 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.

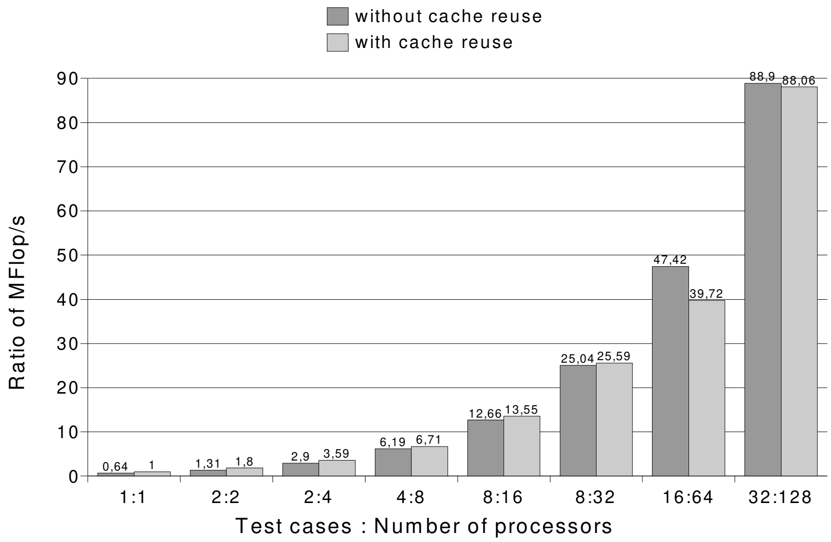

Figure 7: The ratio of MFlop/s for the Cray T3E-900 based on the measurement of test case 1 on 1 processor with switched on cache reuse.

In [6] measurements on further supercomputers like Cray T90, NEC SX-4 and Fujitsu VPP300 are presented. In [3] measurements on the parallel computer IBM SP are presented.

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.

Because of the small memory of 128 MB on the Cray T3E-900 in Jülich we had to increase the number of processors for the test cases 2 and 8 to run the test example up to test case 32. 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. 7 also shows the scalability of the program package LINSOL regarding the computation time.

Linking the own synchronous *message passing* interface instead of MPI only leads to a performance enhancement of 1.4 percent (on 4 processors) till 2.5 percent (on 64 processors) in comparison to the listed performance rates. Using the optimized reduce routine the performance rates stay constant. The minimal performance enhancement in comparison to the benchmarks show that the communication is optimally hidden behind the computation in LINSOL.

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 the parallel computer Cray T3E 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 [4] [6] [8] 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, Lecture Notes in Computer Science*, pages 247–261, Berlin, 1994. Springer.

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

[4] H. Häfner, W. Schönauer, and R. Weiss. The program package LINSOL - basic concepts and realization. *Applied Numerical Mathematics*, 1998. Special Issue on Iterative Methods and Preconditioners, to appear.

[5] 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/>.

[6] 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.

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

[8] 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.