Reliable Computing 1 (2) (1995), pp. 173–187

A reliable linear algebra library for transputer networks

Christian P. Ullrich and Roman Reith

This paper presents a collection of linear algebra subroutines for transputer networks The developed pilot library is intended to form a basis of a complete parallel linear algebra library for validating computations, whose routines will deliver (as accurately as necessary) either the best possible result, or a corresponding inclusion based on controlled rounding and an optimal scalar product

So far, as a first step, we have produced code for interval arithmetic, scalar products and simple vector-matrix operations with maximum accuracy For the solution of triangular systems and the LU decomposition of dense matrices, new versions of classical methods are implemented which allow the application of optimal scalar products as a single, indivisible operation and optimize the overlap of communication and computation. The library also contains routines for computing inclusions of unstructured, dense linear systems of equations.

Network topology dependency is avoided in all numerical routines by the use of general communication routines. This way the user is able to work with different topologies like ring structures, binary trees and hypercubes

Надежная линейно-алгебраическая библиотека для транспьютерных сетей

К. Ульрих, Р. Райт

Представлен набор линейно-алгебраических подпрограмм для транспьютерных сетей Эта пробная библиотека разрабатывалась как основа для полной параллельной линейно-алгебраической библиотеки доказательных вычислений. Входящие в эту библиотеку процедуры будут обеспечивать (с необходимой точностью) либо наилучший возможный результат, либо соответствующее ему включение. Основой для процедур является управляемое округление и оптимальное скалярное произведение.

Пока, в качестве первого шага, нами написаны процедуры для интервальной арифметики, скалярных произведений и простых векторно-матричных операций с максимальной точностью Для решения треугольных систем и LU-разложения плотных матриц реализованы новые версии классических методов, которые позволяют использовать оптимальное скалярное произведение как единую неделимую операцию и оптимизируют одновременное выполнение вычислений и обмена данными в вычислительной сети. Эта библиотека также содержит процедуры для вычисления включений неструктурированных плотных систем линейных уравнений

Во всех численных процедурах мы избегали зависимости от топологии конкретной сети за счет использования обобщенных коммуникационных процедур Поэтому пользователь библиотеки сможет работать с такими различными топологиями, как кольцевые структуры, бинарные деревья и гиперкубы

1. Motivation

As pointed out in [2], many numerical analysts expect not only a well implemented IEEE Arithmetic on a computer, but also the possibility to perform composed calculations like vector and matrix operations in a reliable manner This means that the calculations should deliver the best possible result or a corresponding inclusion. On sequential computers, a wide variety of software for that purpose is offered. from simple library routines to advanced programming environments (see [13]). In contrast, almost no choice is given on parallel computers [3, 4, 12, 14]. The reasons for that situation seem to be:

- The history of parallel computers is too short.
- There are many different architectural concepts (number and coupling of processors).
- The number of different processor topologies is unrestricted.
- The decision on granularity in parallel algorithms cannot be made in general.

In the following sections. a collection of linear algebra subroutines for transputer networks is presented The whole library is designed for validating computations, i.e. the routines should deliver either the best possible result, or a corresponding inclusion based on controlled rounding and an optimal scalar product. The user interfaces of the provided procedures are fully described in [12].

We decided to code all algorithms in OCCAM−2, and (where necessary) in transputer assembler because we want to fully utilize the transputer [7].

2. Basic arithmetic and scalar products

Usual floating-point operations are performed on a T800 transputer with rounding to the nearest floating-point number. The user intending to control the rounding direction is supported by two library routines in OCCAM−2:

```
BOOL, REAL32 FUNCTION IEEE32OP (VAL REAL32 x,
                                VAL INT rd, op, VAL REAL32 y)
BOOL, REAL64 FUNCTION IEEE64OP (VAL REAL64 x,
                                VAL INT rd, op, VAL REAL64 y)
```

where the argument rd specifies the rounding direction and op selects the floating-point operation. Surprisingly the execution time for these operations increases with factor 4 compared to the usual operations, due to the fact that the code is not inlined and the values for rd and op are checked at runtime. A new design of the arithmetic routines reduces this factor to 2 by

- providing one routine for each operation

```
PROC REAL64.ADD.UP      (VAL REAL64 a, b, REAL64 c)
PROC REAL64.ADD.NEAREST (VAL REAL64 a, b, REAL64 c)
PROC REAL64.ADD.DOWN    (VAL REAL64 a, b, REAL64 c)
PROC REAL64.SUB.UP      (VAL REAL64 a, b, REAL64 c)
PROC REAL64.SUB.NEAREST (VAL REAL64 a, b, REAL64 c)
PROC REAL64.SUB.DOWN    (VAL REAL64 a, b, REAL64 c)
PROC REAL64.MUL.UP      (VAL REAL64 a, b, REAL64 c)
PROC REAL64.MUL.NEAREST (VAL REAL64 a, b, REAL64 c)
PROC REAL64.MUL.DOWN    (VAL REAL64 a, b, REAL64 c)
PROC REAL64.DIV.UP      (VAL REAL64 a, b, REAL64 c)
PROC REAL64.DIV.NEAREST (VAL REAL64 a, b, REAL64 c)
PROC REAL64.DIV.DOWN    (VAL REAL64 a, b, REAL64 c)
```

- and by implementing each operation in transputer assembler·

```
PROC REAL64.MUL.DOWN (VAL REAL64 a, b, REAL64 c)
  -- c := a*<b
  -- workspace and registers:
  -- workspace[wsptr+3]:  pointer to c
  -- workspace[wsptr+2]:  pointer to b
  -- workspace[wsptr+1]:  pointer to a
  -- workspace[wsptr]:  return address
  GUY
    LDL 1     -- Areg := pointer to a
    FPLDNLDB -- FAreg := a
    LDL 2     -- Areg := pointer to b
    FPLDNLDB -- FAreg := b
    LDC 5
    FPENTRY   -- set rounding mode to round toward minus infinity
    FPMUL     -- FAreg := a*<b
    LDLP 3    -- Areg := pointer to c
    PFSTNLDP -- c := FAreg
:
```

After this success, we decided to implement real interval arithmetic also in transputer assembler. For the presentation of a real interval $[a]$, we use two floating-point numbers a.inf, a.sup. Since the formulation of the routines is straightforward, we list only the first part of the code for interval addition and the interfaces in the remaining cases.

- $[c] := [a] + [b]$:

```
PROC INTERVAL64.ADD (VAL REAL64 a.inf, a.sup, b.inf, b.sup,
                         REAL64 c.inf, c.sup)
  GUY
    -- c.inf := a.inf +< b.inf
    LDL 1     -- A.reg := pointer to a.inf
    FPLDNLDB -- FAreg := a.inf
    LDL 3     -- Areg := pointer to b.inf
    FPLDNLDB -- FAreg := b.inf
    LDC 5
    FPENTRY  -- set rounding mode to round toward minus infinity
    FPADD    -- FAreg := a.inf +< b.inf
    LDLP 5   -- Areg := pointer to c.inf
    FPSTNLDP -- c.inf := FAreg
    -- c.sup := a.sup +> b.sup
    LDL 2    -- Areg := pointer to a.sup
             ...
:
```

- $[c] := [a] - [b]$:

```
PROC INTERVAL64.SUB (VAL REAL64 a.inf, a.sup, b.inf, b.sup,
                         REAL64 c.inf, c.sup)
```

- $[c] := [a] * [b]$:

```
PROC INTERVAL64.MUL (VAL REAL64 a.inf, a.sup, b.inf, b.sup,
                         REAL64 c.inf, c.sup)
```

- $[c] := [a]/[b]$:

```
PROC INTERVAL64.DIV (VAL REAL64 a.inf, a.sup, b.inf, b.sup,
                         REAL64 c.inf, c.sup)
```

The optimal scalar product[1] plays a central role in almost all self-validating algorithms. For real vectors $x = (x_i)$, $y = (y_i)$ it computes

$$rd\left(\sum_{i=1}^{n} x_i * y_i\right)$$

with one rounding rd applied to the exact value of the scalar product $x * y$. Different approaches to the implementation of this operation are discussed in [2]. The present library provides the long accumulator version implementing the exact scalar product as a fixed point number by an array of 32-bit integers. The whole operation is considered as single, indivisible unit always computed by one processor. This design is preferred since

- distribution of input data to a number of processors and transmission of high accuracy intermediate results would require high communication costs [9]
- in practice scalar products occur during matrix-vector and matrix-matrix operations where the sequential treatment of complete scalar products makes more sense.

[1]The optimal scalar product is also called *maximum-accuracy scalar product*, *maximum quality scalar product*, or *exact scalar product* in literature

All higher numerical algorithms of this library relying on the optimal scalar product are developed to support this design.

The user interface of the optimal scalar product is given by three routines:

```
PROC clear.accus ()
PROC accumulate (VAL [] REAL64 x, y)
PROC round (REAL64 result.up, result.nearest, result.down)
```

where `clear.accus` sets the accumulator to zero, `accumulate` generates the exact value and `round` returns three results by applying three roundings to this value. Note that this can be done nearly without additional costs compared to the production of only one result. A consequence is a speedup of computing an interval inclusion of the scalar product.

Scalar products for interval vectors are implemented analogously.

The effort compared to the conventional scalar product could be worse due to some features of the transputer hardware architecture: first, the CPU of the T800 transputer contains just three general registers, which form a hardware stack. Therefore, it is impossible to hold frequently used variables in a register permanently. Second, unlike the announced T9000 and Motorola 68020, 68030 microprocessors, the T800 transputer does not contain a barrel shifter allowing very fast shift operations. Analysis of the execution time yields that more than 30% of the cycles are needed for load and store operations and additionally, about 15% are needed for shift operations.

	conventional FOR-loop	lower bound using OCCAM−2 library	optimal scalar product
normalized execution times	1	3.1	9.2
number of rounding operations	$2n-1$	$2n-1$	1

Table 1. Normalized execution times and the number of rounding operations of different scalar product implementations ($n = 1000$)

Table 1 shows the normalized execution times and the number of rounding operations required by three scalar product implementations ($n = 1000$):

- a conventional implementation using a simple for-loop with uncontrolled rounding operations and thus unreliable results,
- an implementation using the OCCAM−2 library routine IEEE64OP to determine a lower bound[2] of the true result by performing operations with downwardly directed roundings, and
- an implementation that computes the result of maximum accuracy by applying just one rounding operation to the true value.

[2]without maximum accuracy

3. Communication routines

A multicomputer, like a transputer system, consists of a network of processors, each of which possesses its own local memory. The processors communicate and synchronize with each other by sending messages. However, the OCCAM−2 programming language only supports message passing between two neighboring processors. A realization of non-neighboring communication schemes, which are typical of parallel programs, must take into consideration a number of characteristic hardware and software features: the number of processors, their topology, the number of hardware links, and the one-to-one correspondence between hardware links and software channels. To handle these dependencies in a more or less portable way, some collective communication routines were implemented:

- `distribute`: row-wise distribution of matrices
- `broadcast`: a source node sends a single message (vector) to all other nodes
- `expand`: gather all elements of a distributed vector and make them available to all nodes
- `flow.to.root.receive`, `flow.to.root.send`: a minimal spanning tree is embedded in the network and these two procedures realize an information flow from the leave nodes to the root
- `iteration.control`: hides the necessary communication when performing a convergence check in a parallel iteration

In general, these five communication routines have to be adapted to each new topology. However, some of them are also implemented in a "topology independent" way: they are not implemented by using the send/receive mechanism of OCCAM−2 but by applying one of the other collective communication routines:

communication scheme	was also realized on the basis of ...
`distribute`	`broadcast`
`expand`	`broadcast`
`iteration.control`	`flow.to.root.send`
	`flow.to.root.receive`

Table 2. Hierarchical implementation of communication routines

Besides this hierarchical implementation, all of the mentioned communication routines were directly implemented for a binary tree and a ring topology.

For illustration we list the interfaces of the `broadcast` and `expand` routines:

```
PROC broadcast (VAL INT id,
                        src,
                [] REAL64 vector)
```

```
PROC expand (VAL INT id,
                     dimension,
        VAL [] REAL64 local.vector,
            [] REAL64 global.vector)
```

4. Vector and matrix operations

Our goal is an efficient application of the optimal scalar product in (parallel) vector and matrix operations. Regarding the scalar product as a single, indivisible operation this implies row-oriented algorithms with a row-wise distribution of matrices to the available processors. An invoked routine expects that an input matrix already has been distributed by rows and that just locally stored rows are passed; in contrast, an input vector is always passed completely to the called routine. The computed results are returned in a distributed shape. Each component is of maximum accuracy. Such a component can be the best floating-point approximation, the smallest enclosing interval or the absolute value of the smallest enclosing interval. If the absolute value a of an interval is returned, a so-called symmetric inclusion $[-a, a]$ with $a \geq 0$ was computed. In this case, the interval valued object (vector, matrix) containing the solution is represented only by a point valued object (vector, matrix) which reduces memory requirement. This strategy of using symmetric intervals is especially useful when residuals are to be computed. Since inclusions of residuals are often almost symmetric to the origin, nearly no accuracy is lost in this case.

4.1. Vector and matrix-vector operations allowing simple parallelizations

In this section we discuss vector and matrix-vector operations that allow an interesting parallelization strategy: the problem can be divided into subproblems which have the same structure as the original problem and which can be solved independently from each other, i.e. without transmitting data. We illustrate this strategy by considering a matrix-vector multiplication $M \cdot x$ (see Figure 1). If matrix M is partitioned and distributed by rows, then each scalar product will be computed entirely by the processor owning the corresponding row, and each processor will thus perform a matrix-vector multiplication of reduced problem size.

The advantages of this approach are evident:

- application of the optimal scalar product is supported
- reliable results
- coarse-grain parallelizations
- efficiency
- reusability of sequential code.

Therefore the presented library places at the users' disposal a long series of sequential operations, which can also be used without any changes in a parallel environment. The

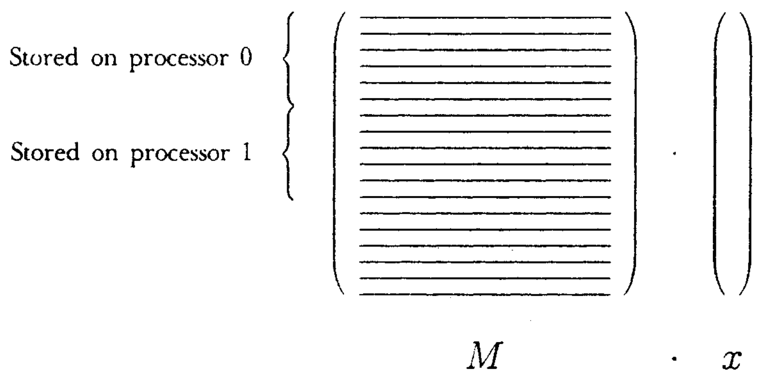

Figure 1. Parallel matrix-vector multiplication

following tables give an overview of some of the corresponding routines using a shorthand notation.

- Sequential vector operations: vectors x, y

inflation:	$[x] \leftarrow [x] + d(x) \cdot [-eps, eps]$
subset:	$[x] \subseteq [y]$ and $[x] \neq [y]$

- Sequential matrix-vector operations: matrices M, R

$M \cdot x$	*real* × *real* → *nearest*
	real × *real* → *interval*
	real × *real* → *sym interval*
	real × *interval* → *interval*
	real × *interval* → *sym interval*
	real × *sym interval* → *sym interval*
	interval × *interval* → *interval*
	sym interval × *sym interval* → *sym interval*
$M \cdot x + y$	*interval* × *interval* + *interval* → *interval*
	sym interval × *sym interval* + *interval* → *sym interval*
$y - M \cdot x$	*real* − *real* × *real* → *nearest*
	real − *real* × *real* → *interval*
	real − *real* × *real* → *sym interval*
	interval − *interval* × *real* → *interval*
	interval − *interval* × *real* → *sym interval*

4.2. Matrix-matrix operations

An important matrix-matrix operation in self-validating algorithms for solving dense linear systems computes inclusions of residual matrices $I - R \cdot M$, where I denotes the identity matrix. Parallelizing such a matrix-matrix multiplication in the same way as the discussed matrix-vector multiplications, i.e. decomposing the operation into independent suboperations that do not need any communication while performing their computation, would waste a lot of memory, because matrix M would have to be hold by each processor completely. Therefore, we apply a more effective strategy. Again, it is assumed that the matrices R and M are distributed among the processors by rows. But now, matrix M flows column-wise through the network during computation. This allows to handle a scalar product as an indivisible operation The communication cost can be masked by overlapping communication and computation.

The librarv provides routines for the following types of problems:

$I - R \cdot M$	*real* − *real* × *real* → *interval*
	real − *real* × *real* → *sym interval*
	real − *real* × *interval* → *interval*
	real − *real* × *interval* → *sym interval*

Sequential routines are included as well as parallel routines with and without overlap of communication and computation. Of course, the routines can easily be modified to compute a matrix-matrix multiplication $R \cdot M$.

5. Solving lower triangular systems

The well-known sequential inclusion methods for lower triangular systems are based on the straightforward application of interval arithmetic: all floating-point operations are replaced by interval operations. The additional evaluation of scalar products with maximum accuracy may lead to sharper inclusions, but in general, especially for very large systems, the results are not guaranteed to maximum accuracy. These difficulties are caused by the recursive nature of the evaluated equations. Nevertheless, we present routines that make extensive use of the optimal scalar product, because this approach seems to be most promising. In particular, sharp inclusions can be expected, when a diagonally dominant matrix is given.

5.1. Sequential implementations

On a serial computer, two classical methods are available to solve triangular systems: the column-oriented and the row-oriented forward substitution. Only the latter allows an effective use of the optimal scalar product. Therefore, a corresponding routine was added to the library.

5.2. Parallel implementations

In the last few years different conventional numerical methods have been developed to solve triangular systems of equations on multicomputers (for example [5] and [6]). Of course, all these methods can be used to compute inclusions by applying simple interval arithmetic. However, they do not efficiently support the optimal scalar product. Therefore, two new versions for solving lower triangular systems $Lx = b$ are developed satisfying these criteria. The so-called *cyclic MSP algorithm*[3] uses ring connectivity and is valid only for wrap mapping of the matrix rows. In this algorithm, a data packet of size $p - 1$, where p denotes the number of available processors, circulates through the network. A significant reduction of the execution time can be reached if this data packet is broken into subsegments that circulate through the network independently (*pipeline version*).

The so-called *MSP broadcast algorithm* can be executed on general networks merely requiring a broadcast routine there and only assuming a row-wise distribution of the matrix rows. The algorithms are described in detail in [12].

To characterize the functionality of the provided library routines in a compact way, the following abbreviations are used:

incl	inclusions of the true result are computed
approx.	approximations of the true result are computed
msp:	optimal scalar product is applied
over:	communication and computation are overlapped
nover:	communication and computation are <u>not</u> overlapped
row pivot:	row pivoting is used, i.e., the maximal element in a column is chosen as a pivot
col pivot:	column pivoting is used, i.e., the maximal element in a row is chosen as a pivot
no pivot:	no pivoting strategy is applied

The library contains the following routines for solving interval valued triangular systems:

- cyclic MSP form: *incl, msp, over*
- pipeline version of the cyclic MSP form: *incl, msp, over*
- pipeline version of the cyclic MSP form: *incl, msp, nover*
- MSP broadcast form: *incl, msp, over*
- MSP broadcast form: *incl, msp, nover*

We list two of the procedure interfaces the first for the MSP broadcast form without overlap and the second for the pipeline version of the cyclic MSP form with overlap of communication and computation.

```
PROC PAR.ILxb.MSP.broadcast (VAL INT id,
                                     dimension,
                                     num.of.stored.rows,
                        VAL [][] REAL64 local.L.inf, local.L.sup,
                        VAL   [] REAL64 local.b.inf, local.b.sup,
                              [] REAL64 x.inf, x.sup)
```

[3]"MSP" stands for maximum-accuracy scalar product.

```
PROC PAR.ILx.b.cyclic.MSP.pipe.over (VAL INT id,
                                             dimension,
                                             num.of.stored.rows,
                                             sigma,
                                 VAL [][] REAL64 local.L.inf, local.L.sup,
                                 VAL    [] REAL64 local.b.inf, local.b.sup,
                                        [] REAL64 x.inf, x.sup)
```

6. Inverting dense matrices

The Gauss-Jordan algorithm is a well-known method for in-place matrix inversion. Different versions of this algorithm are implemented. They compute an approximate inverse by using the available floating-point operations. Therefore, the results are in general not of maximum accuracy. These routines are used in the inclusion algorithm for general linear systems discussed in Section 8.

6.1. Sequential implementations

The library includes a sequential implementation of the Gauss-Jordan algorithm with row pivoting.

6.2. Parallel implementations

The communication of the parallel algorithms is realized by a broadcast routine. Therefore, the adaptation of the algorithms to new network topologies requires just the modification of this communication routine. An input matrix is to be distributed by rows before calling the routines. The decomposed matrices will also be returned distributed by rows.

The routines implemented differ from each other on the pivoting strategy (row pivoting, column pivoting, no pivoting) and whether an overlap of communication and computation is applied or not. In detail, routines based on the following strategies are available:

- *approx, over, row pivot*
- *approx, nover, row pivot*
- *approx, over, col pivot*
- *approx, nover, col pivot*
- *approx, over, no pivot*

Again, we list an interface, to illustrate the use of the routines (Gauss-Jordan algorithm with row pivoting and overlap of communication and computation):

```
PROC PAR.gauss.jordan.row.piv.over (VAL INT id,
                                            dimension,
                                            num.of.stored.rows,
                                        INT numeric.error,
                               [][] REAL64 local.A,
                                 [] INT    swap.cols)
```

7. LU factorization of dense matrices

There are many different ways of organizing LU factorization; all perform the same arithmetic operations, but in a different sequence. Ortega [8] describes 12 such versions, which are called ijk and $ijk2$ forms according to the arrangement of the triply nested loop. The innermost loop performs a scalar product operation only if its loop index is k. In all other cases, it is an AXPY operation. The library contains implementations of the $jik2$ and jik forms, which allow an efficient application of the optimal scalar product. Additionally, some versions of the kij form are included.

7.1. Sequential implementations

The library includes the following types of sequential routines:

- kij form: *approx, col piv*
- jik form: *approx, msp, no piv*
- $jik2$ form: *approx, msp, no piv*

7.2. Parallel implementations

The communication of the parallel algorithms is realized by a broadcast routine. Therefore, the adaptation of the algorithms to new network topologies requires just the modification of this communication routine. An input matrix is to be distributed by rows before calling the routines. The decomposed matrices will also be returned distributed by rows.

7.2.1. kij form

Most implemented routines compute an approximate LU factorization by using the available floating-point operations. Only one routine computes inclusions; all floating-point operations are replaced by interval operations. Therefore, the results are in general not of maximum accuracy.

Routines based on the following strategies are available:

- *approx, over, row pivot*
- *approx, nover, row pivot*

- *approx, over, col pivot*
- *approx, nover, col pivot*
- *approx, over, no pivot*
- *approx, nover, no pivot*
- *incl, over, col pivot*

These different versions allow extensive studies of the effects of pivot strategies and overlap techniques on the efficiency of a parallel algorithm. Such experiences are very important for developing high performance algorithms

7.2.2. *jik* and *jik*2 form

Parallel versions of the *jik* and *jik*2 forms were developed and implemented that allow the application of the optimal scalar product. The library also includes routines based on conventional floating-point operations. All routines compute approximate LU factorizations and thus the results are in general not of maximum accuracy.

In detail, following routines are available:

- *jik*2 form: *approx, msp, over, no pivot*
- *jik*2 form: *approx, msp, nover, no pivot*
- *jik*2 form: *approx, over, no pivot*
- *jik*2 form: *approx, nover, no pivot*
- *jik* form: *approx, msp, over, no pivot*
- *jik* form: *approx, msp, nover, no pivot*

Again, we present one interface just for illustration (parallel *jik*2 form with optimal scalar product and overlap of communication and computation):

```
PROC PAR.LU.decomp.jik2.MSP.over (VAL INT id,
                                          dimension,
                                          num.of.stored.rows,
                                      INT numeric.error,
                               [][] REAL64 local.A)
```

8. Self-validating linear system solver for dense matrices

Computing a highly accurate inclusion for the solution of a general, unstructured linear system $Ax = b$ based on fixed-point methods requires operations that are discussed in previous sections. They include:

- computing an approximate inverse R of the input matrix A
- some basic matrix-vector operations of maximum accuracy
- the inclusion of the residual matrix $I - R \cdot A$ with maximum accuracy
- communication procedures: *broadcast*, *expand*, *iteration.control*

A parallel self-validating linear system solver was build by combining these operations in appropriate manner. Routines for solving point valued and interval valued linear systems are available. Of course, the library also contains the corresponding sequential solvers.

For illustration, we list the interface of the procedure PAR.ILSS that solves a interval valued linear system:

```
PROC PAR.ILSS (VAL INT id,
                       dimension,
                       num.of.stored.rows,
                   INT numeric.error,
             [][] REAL64 local.A.inf, local.A.sup,
           VAL [] REAL64 b.inf, b.sup,
               [] REAL64 x.inf, x.sup)
```

9. Conclusions

A pilot library of basic linear algebra subroutines for transputer networks has been established. The major aim of the work was to gain knowledge and practical experience of the design, implementation, and performance behaviour of parallel self-validating algorithms. Especially the application of the optimal scalar product was investigated in a multicomputer environment. All routines which have been found useful in this context are included to the library. Therefore, its completeness is not claimed but adapation and extensions can be made very easily because the source code can be used by interested research institutes without any charge.

References

[1] Atanassova, L. and Herzberger, J. (eds) *Computer arithmetic and enclosure methods.* Elsevier Science Publisher, North-Holland, Amsterdam, 1992.

[2] Bohlender, G. *What do we need beyond IEEE arithmetic.* In: Ullrich, C. (ed.) "Contributions to Computer Arithmetic and Self-Validating Numerical Methods", IMACS Annals on Computing and Applied Mathematics 7 (1990), J.C. Baltzer AG, Basel, pp. 1–32.

[3] Caprani, O. and Madsen, K. *Performance of an OCCAM/transputer implementation of interval arithmetic.* In: "Abstract for the International Conference on Numerical Analysis with Automatic Result Verification, Lafayette, 1993", p. 12.

[4] Davidenkoff, A. *High accuracy arithmetic on transputers.* In: Kaucher, E., Markov, S., and Mayer, G. (eds) "Computer Arithmetic, Scientific Computation and Mathematical Modelling", IMACS Annals on Computing and Applied Mathematics **12** (1991), J.C. Baltzer AG, Basel, pp. 45–61.

[5] Eisenstat, S. C., Heath, M. T., Henkel, C. S., and Romine, C. H. *Modified cyclic algorithms for solving triangular systems on distributed-memory multiprocessors.* SIAM J. Sci. Stat. Comput. **9** (3) (1988), pp. 589–600.

[6] Heath, M. and Romine, C. *Parallel solution of triangular systems on distributed-memory multiprocessors.* SIAM J. Sci. Stat. Comput. **9** (3) (1988), pp. 558–588.

[7] *Transputer instruction set—a compiler writer's guide.* INMOS Limited, Prentice Hall, 1988.

[8] Ortega, J. M. *The ijk forms of factorization methods.* Parallel Computing **7** (1988), pp. 135–162.

[9] Reith, R. *Scalar products on distributed-memory systems.* In: [1], pp. 129–136.

[10] Reith, R. *Wissenschaftliches Rechnen auf Multicomputern—BLAS-Routinen und die Lösung linearer Gleichungssysteme mit Fehlerkontrolle.* Dissertation, Institut für Informatik, Universität Basel, 1993.

[11] Reith, R. *Description and interfaces of accurate BLAS-routines for transputer networks.* Technical Report 93–5, Institut für Informatik, Universität Basel, 1993.

[12] Reith, R. and Ullrich, C. P. *Large-grain parallelizations of interval algorithms decomposing dense matrices and solving triangular systems on multicomputers.* In: "IMACS/GAMM International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics, SCAN 93, Vienna, September 26–29, 1993".

[13] Ullrich, C. P. *The programming toolbox of the numerical analyst.* In: [1], pp. 69–85.

[14] Wolff von Gudenberg, J. *Implementation of accurate matrix multiplication on the CM*.* In: Albrecht, R., Alefeld, G., and Stetter, H. J. (eds) "Validation Numerics". Computing Suppl. **9** (1993), pp. 287–291.

Received: September 8, 1993
Revised version: June 5, 1994

Institut für Informatik
University of Basel
Mittlere Strasse 142
CH–4056 Basel
Switzerland