ILIAS : a sequential language for parallel matrix computations

Citation for published version (APA):

Document status and date:
Published: 01/01/1993

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Eindhoven University of Technology
Department of Mathematics and Computing Science

ILIAS,
a sequential language for
parallel matrix computations

by
L.D.J.C. Loyens and J. Moonen

93/33

Computing Science Note 93/33
Eindhoven, October 1993
This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. M. Philips
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.
Abstract

The ILIAS system consists of a sequential language for matrix computations, a compiler translating a source program into ILIAS pseudo code and a parallel interpreter for this code. The pseudo code is independent of a target architecture; it merely specifies scalar and matrix computations. We present the ILIAS language and discuss its implementation on a square torus network of transputers. Subscription of matrices causes data-alignment problems, which are solved by redistributions. To reduce redistribution overhead we use a new data distribution called the grid-base distribution. Furthermore, we develop several run-time heuristics that, together with the grid-base distribution, efficiently implement matrix subscription. The feasibility and scalability of the ILIAS system is demonstrated by timing results for two example ILIAS programs, an LU decomposition and a Strassen matrix multiplication, on transputer networks up to 400 processors.

1 Introduction

In this paper, we report on the design and implementation of ILIAS, a new sequential programming language for parallel matrix computations. ILIAS is currently implemented on a square torus network of transputers, but it is suitable for distributed memory architectures in general. Matrix computations can be specified naturally in existing sequential languages like C and Fortran. A parallelizing compiler can identify these computations and generate code to be executed on a multiprocessor system. This can be done automatically, semi-automatically [5, 15, 26] or by using compiler directives [12, 21].

With a parallelizing compiler existing sequential programs can be ported to a parallel architecture by recompilation. The design of a parallelizing compiler for a distributed memory architecture is a difficult and challenging task requiring sophisticated compiler techniques. Until now, most
efforts have been done for shared memory architectures, for example, detection of parallelism in loops (see [6], for some performance figures).

High level operators, like matrix multiplication, have already been introduced in existing sequential languages [7, 18]. Complementary, new constructs more amendable for parallelization have been proposed, for example the forall construct [12]. (See [10, 22] for more general information.) Still, we feel that the current research on a parallelizing system is "burdened" by existing sequential language constructs. Therefore, we start from scratch to define a sequential language for matrix computations in absence of any constructs indicating parallelism. In this language, matrix computations can be specified at a high level. The advantages are twofold. First, programs can be coded compactly in an abstract, mathematical and sequential way. Second, it is relatively easy to recognize program parts suitable for parallelization compared to conventional languages. For example, a matrix multiplication can be expressed as $C := A \cdot B$ (high level) or as (Pascal notation)

$$
\text{for } i := 0 \text{ to } A_m - 1 \text{ do}
\begin{align*}
\text{for } j := 0 \text{ to } B_n - 1 \text{ do} \\
\text{begin} \\
C[i,j] := 0; \\
\text{for } k := 0 \text{ to } A_n - 1 \text{ do} \\
\text{end}
\end{align*}
$$

It should be clear that identifying $C := A \cdot B$ is much easier than identifying the above given program fragment.

This paper is organized as follows. In Section 2 the ILIAS system is sketched. In Section 3 the ILIAS source language is briefly discussed. The implementation of the language on a square torus network of transputers is discussed in Section 4. In Section 5 we give some examples of rephrasing well-known sequential algorithms in ILIAS. Timing results are given as well. Conclusions and suggestions for future work are given in Section 6.

2 The ILIAS system

The ILIAS system consists of three parts: a sequential language, a compiler and a parallel interpreter. The source language is suitable for specifying matrix computations in a MATLAB-like style [8]. A correct source program is translated into a sequence of matrix and scalar computations, which we call the pseudo code of an ILIAS source program. This pseudo code program is sent to a transputer network and executed by a parallel interpreter. The interpreter is built on top of PARPACK [25], a parallel library of basic linear algebra computations for square meshes of transputers.

The ILIAS source language is designed according to the following criteria.

- expressiveness: it is fairly easy to rephrase existing linear algebra programs (written in a conventional language) in the ILIAS language.
- orthogonality: any composition of the basic language primitives is allowed.
- compositionality: it is possible to structure a program into separate building blocks.

Besides functional requirements, the following operational requirements are imposed on the parallel interpreter.
scalability: the execution time and memory usage per processor for an ILIAS program decreases when the number of processors increases.

efficiency: a (sequentially written) ILIAS program, which is interpreted in parallel, is not much slower than an equivalent explicit parallel program.

As mentioned before, a correct ILIAS program is translated into a sequence of pseudo code instructions (which is interpreted in parallel) instead of explicit parallel code. This intermediate code is introduced because of flexibility. It allows the compiler and the interpreter to be modified separately.

The pseudo code we have designed is independent of a target architecture; it merely specifies (in postfix notation) what matrix and scalar computations have to be performed. As a consequence, the pseudo code can be executed on different hardware platforms, thereby increasing the portability of the system.

The independence of a target architecture seems to be a severe restriction for the pseudo code, but it turns out that (for a transputer network) an efficient parallel interpreter can be realized on the basis of this code.

3 The ILIAS source language

In this section, we give an overview of the ILIAS source language. The main features of the language are

- matrix, vector and scalar types, necessary to specify matrix computations.
- generalizations of scalar operators to matrix operators, necessary for expressiveness.
- subscription of matrices, necessary for expressiveness.
- scope, user-defined procedures and functions, resulting in compositionality.
- a library of pre-defined linear algebra routines, resulting in high level operations such as various matrix decompositions and triangular system solvers.

In the remainder we use the following conventions. Matrices are denoted by $A, B, C$, and have dimensions $m \times n$ unless stated otherwise. Scalars are denoted by $a, b, c, d$. The letters $i$ and $j$ denote row and column indices, respectively; they range implicitly over the corresponding matrix dimensions ($0 \leq i < m$ and $0 \leq j < n$). ILIAS keywords and operators are written in typewriter font.

A more detailed description of the language is given next. (A reader may skip this section at first reading and continue with Section 4.)

3.1 Types and variables

ILIAS data types can be divided into the scalar types int and real, and the matrix types fullreal, fullint, rowreal, rowint, colreal and colint. Suffixes int and real refer to the type of the matrix elements. The prefix full is introduced for a general dense $m \times n$ matrix. This is done, because in the future we wish to incorporate features for special matrices like sparse, band or diagonal matrices. Conceptually, row and column vectors (row and col variables) are identified with $1 \times n$ and $m \times 1$ (full) matrices, respectively.
Variables of the type full require two dimensions to be specified, while vector variables (that is row or col variables) only require one dimension to be specified. A dimension specifier may be an arbitrary integer expression as long as it yields a positive number.

Variables are declared in a block of the form \([ \text{declarations statements} ]\), with textual scope between \([ \text{and} ]\). A new scope can be opened at any place where a statement is expected. For example,

\[
\begin{align*}
\text{int } \ m, \ n; \\
\text{m} := 100; \ n := 200; \\
\text{fullreal } A[n, n+m]; \\
\text{rowreal } x[m+n]; \\
\text{colreal } y[m]; \\
\end{align*}
\]

The current implementation of ILIAS lacks explicit Boolean types. Instead, we adopt the C convention and use integers to represent Boolean values. The keywords false and true are identified with integers 0 and 1, respectively.

3.2 Operators

The set of ILIAS operators consists of common scalar operators and generalizations of these scalar operators to matrix operators. In addition to this we have introduced operators for matrix multiplication and matrix transposition.

**Pointwise generalization** ILIAS incorporates the following set \(S\) of binary scalar operators,

\[
S = \{+,-,\ast,/,\backslash,\min,\max,\ =,\ <>,\ >,\ >=,\ <,\ <=,\ \text{and, or}\}.
\]

Operator / denotes integer division when its operands are integer expressions, otherwise it denotes real division. The modulo operator is denoted by \(\backslash\) and can only be applied to positive integer expressions.

A scalar operator \(\oplus\) is a pointwise operator
\[
\text{in a pointwise way by applying } \oplus \text{ to each pair of corresponding matrix elements or to each matrix element and a scalar. Formally,}
\]

\[
\begin{align*}
(A \oplus B)_{ij} &= A_{ij} \oplus B_{ij} \\
(a \oplus B)_{ij} &= a \oplus B_{ij}
\end{align*}
\]

In ILIAS, a pointwise operator \(\oplus\) is denoted by the same symbol as its scalar version \(\oplus\). When this is ambiguous, the operator symbol is prefixed with a point symbol (.). For instance, \(A \ast B\) denotes conventional matrix multiplication and \(A \cdot B\) denotes pointwise matrix multiplication. All pointwise relational and logical operators are prefixed with a point because these operators are also generalized in a Boolean way.

**Boolean generalization** The Boolean generalization of relational and logical scalar operators from the set \(R\),

\[
R = \{<> , >, >=, <, <=,\ \text{and, or}\},
\]
is defined by

\[ A \oplus B \equiv \forall_{i,j} A_{ij} \oplus B_{ij} \quad (2) \]
\[ a \oplus B \equiv \forall_{i,j} a \oplus B_{ij} \]

A pointwise relational or logical operator returns a Boolean (integer) matrix, when applied to matrix operands. A Boolean generalized operator, however, returns a Boolean scalar, i.e., true or false. For example, \( A < B \) returns true if all elements of \( A \) are smaller than the corresponding elements of \( B \), but \( A < B \) returns a Boolean matrix whose elements are obtained from comparing the corresponding elements of \( A \) and \( B \).

Besides binary operators, ILIAS incorporates the unary operators -., not, .not and . The first three operators are written prefix; the operator . is written postfix: \( A^t \) returns the transpose of \( A \).

### 3.3 Standard functions and quantifiers

ILIAS incorporates a set of standard scalar functions, such as sqrt, sin, abs, etc., which have been generalized to matrix functions in a pointwise way. Other standard functions include the dimension functions rdim and cdim, and the type conversion functions toint, toreal and toscal.

For an \( m \times n \) matrix \( A \), \( \text{rdim}(A) \) returns \( m \) and \( \text{cdim}(A) \) returns \( n \). These functions are useful when the dimensions of a matrix are unknown (e.g., when the matrix is a formal parameter of a function or procedure).

Besides some natural implicit type conversion rules (like in C), we have introduced functions for explicit type conversion. For example, \( \text{toint}(A) \), for a real matrix \( A \), returns an integer matrix with truncated real values. The function toscal converts a \( 1 \times 1 \) matrix or a vector of length 1 to a scalar.

Additional expressiveness is introduced by quantifiers. A quantifier is an unary matrix function associated with a scalar operator \( \oplus \) from the set \{ +, *, \text{min}, \text{max}, \text{and}, \text{or} \}. The associated quantifier \( \mathcal{Q}_\oplus \) is defined by

\[ \mathcal{Q}_\oplus(A) = \bigoplus_{i,j} A_{ij} \quad (3) \]

For example, the quantifier associated with + is matsum. \( \text{matsum}(A) \) returns the sum of all matrix elements \( A_{ij} \).

### 3.4 Subscription

An important feature of ILIAS is the subscription of matrix variables. Submatrices are selected by consecutive row and column ranges. A consecutive part is denoted by \( a : b \) specifying row (or column) numbers \( a \) up to and including \( b \). For example, \( A[a : b, c : d] \) specifies a matrix \( B \) with dimensions \( b - a + 1 \times d - c + 1 \) and \( B_{ij} = A_{a+i, c+j} \).

Some abbreviations for ranges can be used. The range \( a : a \) may be abbreviated to \( a \); the range \( _ : _ \) : \( a \) specifies all numbers up to and including \( a \) and \( a : _ \) specifies all numbers starting from \( a \) and within the matrix bounds.

Subscripted matrices and vectors may occur in an expression and in the left-hand side of an assignment. This considerably enriches the expressiveness of language. For instance, a rank-one

---

1 This operator has to be binary, commutative and associative.
update of the right-lower submatrix of \(A\) starting in element \(A_{kk}\) by column and row \(k\) of \(A\) is expressed as

\[
A[k:\_ , k:\_] := A[k:\_ , k:\_] + A[\_ , k] * A[k , \_]
\]

Subscription is restricted to consecutive parts of matrix and vector \textit{variables}. Hence, subscription of arbitrary matrix expressions, like \((A + B)[2,3]\), and subscription with non-unit strides are not implemented yet.

3.5 Statements

Assignment An assignment has syntax

\[
\text{expression} := \text{expression}
\]

where \text{expression} is a variable with an optional subscript. A scalar expression can be assigned to a matrix variable by assigning the expression to each matrix element. A matrix can only be assigned to a scalar if its dimensions are \(1 \times 1\). If a matrix expression is assigned to a (subscripted) matrix variable, the corresponding dimensions must be equal.

Selection and repetition Currently, IILIAS contains one selection construct and one repetition construct. The selection construct has syntax

\[
\text{if expr then statements fi} \quad \text{or} \quad \text{if expr then statements else statements fi}
\]

The first selection statement is a \text{skip} when its guard is \text{false}. Guards must yield an integer scalar value. A loop is expressed by

\[
\text{while expr do statements od}
\]

I/O I/O in IILIAS is kept simple by offering the two instructions \text{read} and \text{write}. \text{read} requires a filename and a list of variables, \text{write} requires optionally an append symbol (indicating whether the output should be appended or not), a filename, and a list of expressions.

User-defined procedures and functions Procedures and functions can be declared at any place in the declaration part of a block. The syntax is

\[
\text{proc proccname (formal parameters)} \quad \text{or} \quad \text{func type funcname (formal parameters)}
\]

\[
\text{declarations} \quad \text{or} \quad \text{declarations}
\]

\[
\text{statements} \quad \text{or} \quad \text{statements}
\]

\[
\text{cor} \quad \text{cnuf}
\]

The formal parameters together with the enclosing parentheses are optional. A parameter can have one of two kinds: \text{value} or \text{value-result}. Default is the value parameter mechanism; the keyword \text{var} specifies \text{value-result} parameters (like in Pascal). Functions return values by the statement \text{return expression}.

Matrix parameters and matrix function results do not require a specification of their dimensions. This way, it is possible to write generic procedures and functions, which may take arbitrary sized matrices as arguments. For example, a Householder vector [11] can be computed by the function \text{house as}
func colreal house(colreal x) { rdim(x) > 1, x <> 0 }
    real mu, beta;
    mu := norm2(x);
    if x[0] > 0 then
        beta := x[0] + mu
    else
        beta := x[0] - mu
    fi;
    x[0] := 1;
    x[1:] := x[1:] / beta;
    return x
end

In the example, a library function norm2 is used and the column vector x is scaled by the pointwise operator /. The function house may be called with a column vector as argument, e.g., house(A[.., b]).

3.6 Library

In the previous subsections, we described the ILIAS language. In addition to this, there is an ILIAS library, which contains high level linear algebra routines. This library corresponds to PARPACK, the parallel library.

The library includes several procedures for matrix decomposition (LU, Cholesky, QR) and various triangular system solvers. Besides this, there are library functions for inner product and norm computations of vectors, functions for permuting matrix rows and columns, and search functions like findmax, which locates a maximal element of a matrix.

4 Interpreting on a processor network

In this section, we discuss the interpretation of pseudo code by a processor network with distributed memory, such as transputer networks.

4.1 Run-time model

The parallel interpreter performs scalar and matrix computations on a Q x Q torus network. Each processor is identified by an ordered pair (s, t), 0 ≤ s, t < Q, and has a part of each matrix in its local memory. Parallelism is achieved by letting each processor operate on its own matrix part as much as possible (data parallelism).

Both PARPACK and the pseudo code are replicated over the processor network, i.e., stored in each processor's local memory. An outline of the interpreter for processor (s, t) is

```
do "not terminated"
    "fetch next instruction c"
    "contribution of processor (s, t) to execution of c"
od
```

The interpreter has to deal with two kinds of objects: scalars and matrices. A matrix is

---

2 Row vectors are implemented as 1 x n matrices, column vectors are implemented as m x 1 matrices.
distributed over the processor network, while a scalar is replicated over the processor network. Scalar instructions are performed *asynchronously* on all processors, while matrix instructions lead to parallel PARPACK calls. Because each processor is presented the same pseudo code, all processors participate in the same matrix computation.

We have chosen to perform scalar operations on all processors instead of on a particular processor. This way, scalar values represent global information and operations involving both scalars and matrices can be performed immediately, without the need for broadcasting scalar values from a particular processor first. Another possibility is to perform scalar operations on a separate processor, which does not participate in matrix operations. This allows for some matrix and scalar operations to be performed in parallel, but also leads to broadcasting of scalar results. Because we expect matrix operations to dominate the execution time, the gain of this additional kind of parallelism will be relatively small. We therefore expect the highest performance for an implementation which replicates scalars.

4.2 The grid distribution

In PARPACK, matrices are distributed over \( p = Q^2 \) processors according to the grid distribution [24]. Matrix \( A \) is grid distributed if element \( A_{ij} \) is assigned to processor \((i \mod Q, j \mod Q)\).

The grid distribution, also called *scattered square decomposition* [9] or *cyclic storage* [13], is the standard distribution in PARPACK for dense matrices, because of its good load balance properties and low communication costs for a wide range of linear algebra computations. These include matrix multiplication [16], LU decomposition [2], Cholesky factorization [17] and triangular system solving [3].

The implementation of declarations, statements, user-defined functions, recursion etc. involves standard compiler techniques [1] and is not discussed here. Subscription of matrices causes data-alignment problems and is discussed next.

4.3 Implementing matrix subscription

A subscripted matrix \( A[a:b, c:d] \) specifies a matrix \( B \) with \( B_{ij} = A_{a+i, c+j} \). Matrix \( A \) is grid distributed, so element \( B_{ij} \) is assigned to processor \((i \mod Q, j \mod Q)\). We call this distribution of matrix \( B \) a grid-base\((a,c)\) distribution.

On a \( Q \times Q \) torus there are \( Q^2 \) different grid-base distributions. The *location* of a matrix is the processor to which matrix element \((0,0)\) is assigned. It characterizes a particular grid-base distribution. Two matrices are *aligned* if they have the same location.

The grid and grid-base distribution are visualized in Figure 1. The grey-shading of a matrix element denotes the processor to which the element is assigned. The figure shows that subscription \([2:6,3:5]\) defines a grid-base\((2,3)\) or grid-base\((0,1)\) matrix (on a \( 2 \times 2 \) processor network).

In our implementation, a matrix \( A \) has four global attributes: \( A.m, A.n, A.ib \) and \( A.jb \), specifying a matrix with dimensions \( m \times n \) and distribution grid-base\((ib,jb)\). A subscripted matrix \( A[a:b, c:d] \) can be viewed as a matrix \( B \) with

\[
(B.m, B.n, B.ib, B.jb) = (b - a + 1, d - c + 1, a \mod Q, c \mod Q)
\]

The attributes \( ib \) and \( jb \) of an unsubscripted matrix are set to zero, because grid = grid-base\((0,0)\).
4.3.1 Initial approach

There are two kinds of subscription in the ILIAS source language, namely subscription of a matrix in an expression and subscription of a matrix in the left-hand side of an assignment. If an operation on a subscribed matrix is to be performed, we redistribute this grid-base matrix to a grid matrix before calling PARPACK, since PARPACK requires grid distributed matrices. In case of a subscribed left-hand side, we redistribute (after evaluation) the grid right-hand side according to the left-hand side distribution. Redistributing matrix $A$ to grid-base$(d_i, d_j)$ requires that all $A_{ij}$ are routed from processor $(A_{ib+i} \mod Q, (A_{jb+j} \mod Q)$ to processor $((di+i) \mod Q, (dj+j) \mod Q)$. This is also called redistributing $A$ to location $(di, dj)$.

4.3.2 Improvement 1

Consider an expression $A[a : b, c : d] \oplus B[a' : b', c' : d']$, where $\oplus$ denotes a pointwise operator. According to the above scheme, every subscription leads to a redistribution. If $A[a : b, c : d]$ and $B[a' : b', c' : d']$ are aligned then all corresponding $A$ and $B$ elements are assigned to the same processor. By generalizing the procedures implementing pointwise operations, the result can be computed without redistributions and therefore without communication. (Because the grid and grid-base distribution have similar load balance properties, this generalization does not affect the efficiency). If $A[a : b, c : d]$ and $B[a' : b', c' : d']$ are not aligned then all corresponding $A$ and $B$ elements are assigned to different processors. In that case, communication is inevitable and both matrices are redistributed to grid.

We have also generalized the remaining PARPACK procedures as much as possible, without losing efficiency. All unary operations have been generalized for arbitrary grid-base matrices; no redistributions are required. A binary pointwise operation requires aligned operands, as discussed above, and returns a matrix with the same distribution as the operand matrices. For
the remaining operations, the precondition aligned is sometimes too strong. As an example, we
discuss the generalization of matrix multiplication.

grid version The PARPACK matrix multiplication $C = A \ast B$, for $A : m \times n$ and
$B : n \times q$, is based on the following sequential algorithm, sometimes called the $kij$- or outer product-form [20].

For $i, j: 0 \leq i < m \wedge 0 \leq j < q$

$\rightarrow C_{ij} := 0$

for $k: 0 \leq k < n$

$\rightarrow$ for $i, j: 0 \leq i < m \wedge 0 \leq j < q$

$\rightarrow C_{ij} := C_{ij} + A_{ik} \ast B_{kj}$

This algorithm is parallelized as follows. Each processor $(s, t)$ loops over $k$ and updates its
local $C_{ij}$. In step $k$, processor $(s, t)$ needs values $A_{ik}$, for $i \mod Q = s$ and values $B_{kj}$, for
$j \mod Q = t$. Hence, we have to replicate matrix column $A_{\ast k}$, which is initially distributed
over processor column $k \mod Q$, over each processor column. Similarly, we have to replicate
matrix row $B_{k \ast}$, over each processor row. We give the algorithm for a processor $(s, t)$, where
$locals.C.s.t$ denotes the set of indices of the local $C$ elements of $(s, t)$.

For $i, j: (i, j) \in locals.C.s.t$

$\rightarrow C_{ij} := 0$

for $k: 0 \leq k < n$

"replicate matrix column $A_{\ast k}$ over each processor column, store in array $Acol$";
"replicate matrix row $B_{k \ast}$ over each processor row, store in array $Brow$";

for $i, j: (i, j) \in locals.C.s.t$

$\rightarrow C_{ij} := C_{ij} + Acol_i \ast Brow_j$

grid-base version Here, matrices $A$ and $B$ are $grid-base(A.ib, A.jb)$ and $grid-base(B.ib, B.jb)$
distributed, respectively. We try to stay as close as possible to the original grid algorithm. In step
$k$, we therefore replicate matrix column $A_{\ast k}$ over each processor column and we replicate matrix
row $B_{k \ast}$ over each processor row. Now, $A_{ik}$ is available on processor row $(A.ib + i) \mod Q$ and
$B_{kj}$ is available on processor column $(B.jb + j) \mod Q$.

Next, observe that the products $A_{ik} \ast B_{kj}$, for $0 \leq k < n$, can be computed at processor
$((A.ib + i) \mod Q, (B.jb + j) \mod Q)$. Hence, no redistributions are necessary if we compute
the result matrix $C$ as a $grid-base(A.ib, B.jb)$ matrix. We have arrived at an algorithm for
arbitrary $grid-base$ matrices, which does not require redistributions and which is equally efficient
as its grid version. The complexity is $O(nm^2) + O(n^{3/2}d)$. The grid and $grid-base$ distributions are very suitable for ILIAS subscripts, because an arbitrary
subscripted matrix is just another $grid-base$ matrix. If the original matrices would be $block$
distributed (i.e., a consecutive submatrix is assigned to each processor) then an arbitrary subscripted matrix would, in general, not be evenly distributed over the processors. Consequently,
this would lead to load imbalance.
4.3.3 Improvement 2

In the previous method, assigning a *grid-base* right-hand side to an unsubscripted (and therefore *grid*) left-hand side involves redistributing the right-hand side to *grid*. The idea is to avoid such a redistribution, by letting the left-hand side inherit the right-hand side distribution. This way, an assignment to an unsubscribed left-hand side is performed without communication. Let $A$ and $B$ be *grid* matrices. An assignment $A := B[a : b, c : d]$ is performed without communication by changing the distribution of matrix $A$ to *grid-base*$(a, c)$. (The distribution of $B$ remains the same.)

Hence, matrix distributions can change run-time. The initial distribution (for all matrices) is *grid* or *grid-base*(0,0). Now, matrices with syntactically equal subscripts are not always aligned because of earlier redistributions.

The distribution of the right-hand side is only inherited by an unsubscripted left-hand side. Otherwise, we introduce different distributions for different parts of a single matrix. This complicates computations and destroys the good load balance properties of the *grid-base* distribution.

4.3.4 Improvement 3

This method improves on the previous ones by using heuristics for each operation to reduce redistribution overhead.

Consider a pointwise operation $A \oplus B$, where $A$ and $B$ are not aligned. In the previous methods, both $A$ and $B$ are always redistributed to *grid* (location (0,0)). Instead of redistributing both matrices to (0,0), we can also choose a more suitable location, redistribute the operands to this location and compute the result there. If we consider $A \oplus B$ in isolation then the location of $A$ and the location of $B$ are equally suitable, because $A$ and $B$ have equal dimensions. A particular choice, however, affects the location of the result. Hence, to minimize redistribution overhead, we must inspect parts of the context of the operation (the whole expression tree and the left-hand side of the expression). In general, we must inspect the whole program, since the location of a right-hand side expression may determine the distribution of a left-hand side matrix, which in turn may be used in other expressions. The problem becomes more complicated if we include operators that take matrices of different dimensions (e.g., matrix multiplication).

In [4] several algorithms are described that, given an expression tree, a cost function and a metric, compute optimal locations for the intermediate results. These algorithms are quite general, but assume that the location and dimensions of each matrix is known compile-time. The related problem of determining optimal initial allocation of matrices is discussed in [14].

In our case, a compile-time analysis of the expression tree to determine optimal locations is not applicable, because the dimensions, subscript values and distributions of matrices are mostly unknown. But, since we are designing the interpreter as well, we can postpone the choice of locations until run-time when more information is available. As a consequence, the pseudo code can remain independent of the network topology and the number of processors.

In case of data-alignment problems, the interpreter chooses locations for intermediate results by several simple heuristics. These heuristics involve the operator, the operands and a possible *target-location*. Consider minimizing the cost of performing an assignment of the form $C := A \oplus B$, where $A$, $B$ and $C$ may be subscripted and $\oplus$ is a binary matrix operator. If $C$ is unsubscripted then its distribution is irrelevant for the chosen location, because $C$ will inherit the right-hand side distribution anyhow. If $C$ is subscripted then, after computation, we need an extra redistribution of the result to the location of $C$. In this case, the location of $C$ is a *target-location* for $A \oplus B$. We generalize this notion of *target-location* from assignments $C := A \oplus B$ to general assignments $C := \text{expression}$ in the following way.
An operation $A \oplus B$ has its left-hand side as a target-location if its left-hand side is subscripted and $A \oplus B$ does not occur in an argument of a function call.

For example, in program fragment $A > B$ then, operation $A > B$ has no target-location. Also in $A[1, 1] := \text{matsum}(B+C)$ operation $B+C$ has no target-location. In $A[a, b] := B[a, b] + C[c, d]$, the target-location is $((A.ib + a) \mod Q, (A.jb + b) \mod Q)$.

If there is a data-alignment problem with no target-location then we redistribute the smallest operand to the location of largest operand (if necessary). Otherwise, the location of the intermediate result is determined by an heuristic involving the specific operator, the operands and the target-location. The heuristics for pointwise operations and matrix multiplication are discussed next in more detail.

**Pointwise heuristic** Consider $A \oplus B$ with target location $(t_i, t_j)$. Let $C$ record the result of the operation. The chosen location $(C.ib, C.jb)$ minimizes

$$Cost(A, C.ib, C.jb) + Cost(B, C.ib, C.jb) + Cost(C, t_i, t_j)$$

where $Cost(A, di, dj)$ is the cost for redistributing $A$ to $(di, dj)$. The first two terms represent the cost of redistributing the operands before computation. The last term represents the cost of redistributing the result to $(t_i, t_j)$. (Here we assume that two redistributions are not performed in parallel, as is the case in our implementation.)

In our case, $C.ib$ and $C.jb$ can be determined separately. Furthermore, in order to determine $C.ib$, we only have to compare the costs associated with choices $A.ib, B.ib$ and $t_i$ [19]. Coordinate $C.jb$ can be determined similarly. Hence, a location that minimizes expression (4) can be determined in constant time.

**Matrix multiplication heuristic** Recall that a matrix multiplication $A \ast B$ returns a grid-base$(A.ib, B.jb)$ matrix with dimensions $A.m \times B.n$. If target-location $(t_i, t_j) \neq (A.ib, B.jb)$ then it might be better to redistribute $A$ and $B$ before multiplication in order to produce the result $C$ at the desired location. (Note that the redistributions of $A$ and $B$ are not necessary, because matrix multiplication does not require aligned operands. It may be beneficial to redistribute anyhow, because the multiplication may return a larger result matrix than the operand matrices.)

The chosen location $(C.ib, C.jb)$ minimizes

$$Cost(A, C.ib, A.jb) + Cost(B, B.ib, C.jb) + Cost(C, t_i, t_j)$$

The first two terms represent the cost of the "cheapest" redistributions that are necessary to obtain a grid-base$(C.ib, C.jb)$ result. The last term represents the cost of redistributing the result to $(t_i, t_j)$. In our case, $C.ib$ and $C.jb$ that minimize expression (5) can be determined in constant time [19] by

$$C.ib = \begin{cases} A.ib & \text{if } \text{size}(A) \geq \text{size}(C) \\ t_i & \text{otherwise} \end{cases}, \quad C.jb = \begin{cases} B.jb & \text{if } \text{size}(B) \geq \text{size}(C) \\ t_j & \text{otherwise} \end{cases}$$

where $\text{size}(A)$ is the maximum number of local matrix elements of $A$. If the operands $A$ and $B$ are larger than the result $C$ then no redistributions are performed. Otherwise, $A, B$ or both $A$ and $B$ are redistributed before multiplication.
Other operations  The heuristics for the remaining operations do not involve a target-location, because either the result is a scalar (e.g., a Boolean operation) or the result is a smaller matrix than one of the operand matrices (e.g., a triangular system solver, which takes a matrix and a column vector, and returns a column vector). In these cases, the smallest operands are always redistributed to the location of the largest operand (if necessary).

Our heuristics have been designed to be optimal for simple assignments of the form \( A := B \oplus C \), where \( \oplus \) is a binary matrix operator and \( A, B \) and \( C \) may be subscripted. In practice, however, they also optimal for assignments such as \( A := A + B \cdot C \).

In an assignment, we use the location of the subscripted left-hand side as the only target-location for all operations in the right-hand side. We could further improve on this by choosing different target-locations for different operations within the same expression.

5 Examples and timing results

In this section, we give two examples of ILIAS formulations of well-known linear algebra algorithms. We discuss an LU decomposition and a Strassen matrix multiplication. Timing results for these examples are given as well.

5.1 LU decomposition

A parallel algorithm that performs LU decomposition has already been incorporated in PARPACK and is available in ILIAS by library procedure ludeco. Programming an LU decomposition in terms of ILIAS operators, however, demonstrates nicely the power of matrix subscription. LU decomposition (with partial pivoting) of a matrix \( A \) results in a permutation matrix \( P \), a unit lower triangular matrix \( L \) and an upper triangular matrix \( U \) with \( PA = LU \). We give an ILIAS formulation in which matrix \( A \) is permuted directly and overwritten by \( L \) and \( U \). The diagonal of \( A \) is overwritten by the diagonal of \( U \). All diagonal elements of \( L \) have value \( 1.0 \) and are, therefore, not stored. The permutation is recorded in a row vector \( P \).

```plaintext
proc LUDecomposition(var fullreal A, var rowint P)
int i, j, k;
real pivot, abspivot;
k := 0;
while k < cdim(P) do P[k] := k; k := k + 1 od;
k := 0;
while k < rdim(A) - 1 do
findmax(abs(A[k:_,k]), abspivot, i, j);
pivot := A[k+i,k];
swaprow(A, k+i, k);
swapcol(P, k+i, k);
A[k+1:_,k] := A[k+1:_,k] / pivot;
A[k+1:_,k+1:] := A[k+1:_,k+1:] - A[k+1:_,k] * A[k,k+1:];
k := k + 1
od;
corp
```

First, row \( P \) is initialized to the identity permutation. In step \( k \), the pivot row is determined by \texttt{findmax}. After the pivot search, matrix \( A \) and row \( P \) are permuted by \texttt{swaprow} and \texttt{swapcol},
respectively. Column $k$ is scaled by $A[k+1:_,k] := A[k+1:_,k] / \text{pivot}$ and the update of the submatrix is formulated as $A[k+1:_,k+1:] := A[k+1:_,k+1:] - A[k+1:_,k] * A[k,k+1:]$.

5.2 Strassen multiplication

Strassen multiplication [23] is a divide and conquer method for matrix multiplication. Consider the following 2-by-2 block matrix multiplication, where each block is square.

$$
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix} =
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
$$

Strassen has shown that matrix $C$ can be computed by 18 matrix additions and 7 matrix multiplications of the half-sized matrices $A_{ij}$ and $B_{ij}$. (The straightforward method requires 8 matrix additions and 8 matrix multiplications of the half-sized matrices.) Furthermore, if $A$ and $B$ have dimensions $n \times n$, where $n$ is a power of 2, we can apply the Strassen method recursively on the multiplication of the half-sized matrices. As the matrices become small, it may be beneficial to switch over to conventional matrix multiplication. The complexity of the Strassen method is $O(n^{\log_27}) = O(n^{2.807})$, compared to $O(n^3)$ for the conventional matrix multiplication.

The Strassen multiplication is implemented in IILAS by a recursive function $\text{Strass}$. In the program text, $n \leq n_{\text{min}}$ specifies the switching criterion to use conventional matrix multiplication. To increase readability, we have used some auxiliary variables.

```plaintext
func fullreal Strass(fullreal A, B; int nmin)
int m, n;

n := rdim(A);
if n <= nmin then return A*B
else m := (n / 2);
fi;

| fullreal p1[m,m], p2[m,m], p3[m,m], p4[m,m], p5[m,m], p6[m,m], p7[m,m];
| fullreal A11[m,m], A12[m,m], A21[m,m], A22[m,m];
| fullreal B11[m,m], B12[m,m], B21[m,m], B22[m,m];
| fullreal C[n,n];

A11 := A[0:m-1,0:m-1]; A12 := A[0:m-1,m:];
A21 := A[m:,0:m-1]; A22 := A[m:,m:];
B11 := B[0:m-1,0:m-1]; B12 := B[0:m-1,m:];
B21 := B[m:,0:m-1]; B22 := B[m:,m:];

p1 := Strass(A11 + A22, B11 + B22, nmin);
p2 := Strass(A21 + A22, B11, nmin);
p3 := Strass(A11, B12 - B22, nmin);
p4 := Strass(A22, B21 - B11, nmin);
p5 := Strass(A11 + A12, B22, nmin);
p6 := Strass(A21 - A11, B11 + B12, nmin);
p7 := Strass(A12 - A22, B21 + B22, nmin);

C[0:m-1,0:m-1] := p1 + p4 - p5 + p7;
C[0:m-1,m:] := p3 + p5;
C[m:,0:m-1] := p2 + p4;
```

14
Implementing Strassen multiplication in ILIAS requires little programming effort and skill, because the ILIAS formulation is very close to the mathematical formulation. Programming an explicit parallel Strassen multiplication (based on the grid distribution) would certainly be not as trivial as this. Other linear algebra algorithms which have been implemented in ILIAS include a Conjugate Gradient Solver [20] and a reduction to bi-diagonal form [11].

5.3 Timing results

In this section, we give some timing results for the LU decomposition and the Strassen multiplication. We discuss the scalability and efficiency, as described in Section 2.

5.3.1 Scalability

The fixed-size speedup $S.p.n$ of an ILIAS program running on $p$ processors with problem size $n$ is defined as $S.p.n = \frac{T.p.n}{T.p.n_0}$, where $T.p.n$ denotes the time for executing the pseudo code of an ILIAS program with problem size $n$ on $p$ processors. The number of processors ($p = 1, 25, 64, 100, 144, 196, 256, 328, 400$) is plotted against the fixed-size speedup of the ILIAS program, for various problem sizes.

![Fixed-size speedup for the LU decomposition](image)

**LU decomposition** Due to the grid-base distribution and the matrix multiplication heuristic, procedure LU_decomposition does not require redistributions. Timings have been obtained for double precision (64-bit) arithmetic. Figure 2 demonstrates the scalability of the LU decomposition, as long as the problem size stays large enough. The maximal speedup obtained is about 150 for case $n = 800$ on 400 processors.
Because of memory limitations for the case $p = 1$, we have tested only relative small problems. Hence, the ratio $s_{p,n}$, for fixed $n$, decreases for increasing $p$.

The fact that better fixed-size speedups are obtained for large problems is explained as follows. The complexity for the LU decomposition is $O(n^3_p) + O(n^3_{\sqrt{p}})$. If $n$ is increased for fixed $p$, the $O(n^3_p)$ term will dominate the overall complexity and the fixed-size speedup factor will approximate $p$.

**Strassen multiplication** Function *Strass* uses subscription and, in general, redistributions are required to compute the actual parameters for each recursive function call. The original matrices are $n \times n$, where $n$ is a power of 2. If $Q$, the dimension of the torus, is also a power of 2, and $n_{\text{min}} \geq Q$, then no redistributions are required. All subscripted matrices are aligned and due to the grid-base distribution, all matrix additions are performed without communication. The redistribution overhead in this case is zero.

Because of memory limitations for case $p = 1$, parameter $n_{\text{min}}$ was set to $n/2$ and the recursion was only one level deep. Timings were obtained for single precision (32-bit) arithmetic.

![Figure 3: fixed-size speedup for the Strassen multiplication](image)

Similar to the LU decomposition, Figure 3 demonstrates the scalability of *Strass*, as long as the problem size stays large enough. The maximal speedup obtained is about 135 for case $n = 512$ on 400 processors. Indeed, a better performance is observed for cases where $Q$ is a power of 2 ($p = 64, 256$). The redistribution overhead is not dominant. We have observed that for $n \geq 512$, case $p = 400$ (with redistributions) outperforms case $p = 256$ (no redistributions).

**5.3.2 Efficiency**

How efficient is an ILIAS program compared to an “equivalent” explicit parallel program? In order to answer this question, we discuss the LU decomposition and the Strassen multiplication.

**LU decomposition** We have compared the execution time of the LU decomposition, programmed in ILIAS, as given in Section 5.1 ($T_{\text{ILIAS}}$), to the time of an ILIAS program consisting
of a single library call ludeco(A, p) ($T_{\text{parpack}}$). The latter program is executed by a parallel LU decomposition algorithm of PARPACK. The ratio $\frac{T_{\text{parpack}}}{T_{\text{ilias}}}$ is plotted against problem size $n$, for $p = 400$. Figure 4 shows that for large $n$, the sequential programmed LU decomposition obtains up to 0.55 of the speed of the parallel programmed LU decomposition. For large $n$, the update phase dominates the execution time of the whole decomposition. The ILIAS update requires two traversals through the matrix data structure. One to perform the multiplication and one to perform the addition. The PARPACK update is implemented by a single traversal through the matrix data structure, where in each step an addition and a multiplication is performed.

**Strassen multiplication** We do not have an explicit parallel algorithm that performs Strassen multiplication, so an exact measure for the efficiency of Strass cannot be given. Nevertheless, we can compare the execution time of the sequential programmed function $\text{Strass}$ ($T_{\text{strass}}$) to the execution time of the parallel programmed conventional matrix multiplication of PARPACK ($T_{\text{matmul}}$). This should give us an idea of when to use the user-defined Strass function instead of the * operator.

In the sequential case, Strassen multiplication becomes more efficient than conventional multiplication for large matrices, say $n \geq 512$. In the parallel case, the matrix is distributed over processors and each processor operates on his own matrix, which is a factor $p$ smaller. Hence, we expect to break even for larger $n$ in the parallel case.

The ratio $\frac{T_{\text{matmul}}}{T_{\text{strass}}}$ is plotted against problem size $n$, for $p = 400$. Again, parameter $n_{\text{min}}$ was set to $n/2$. We see that for the largest problem we could test ($n = 4096$), the sequential programmed function $\text{Strass}$ breaks even with the parallel programmed conventional matrix multiplication. In the special case $p = 256$ (no redistributions) we have observed that for $n = 2048$ Strass is already slightly faster than the conventional multiplication of PARPACK.

![Figure 4: ILIAS-ludeco vs PARPACK-ludeco](image-url)
Figure 5: ILIAS-strass versus PARPACK-matmul

6 Future work and conclusions

6.1 Future work

ILIAS can be functionally extended by new data types and new library routines. For example, a matrix type sparse together with all non-library operations can be added. New parallel algorithms added to PARPACK can also be made available in ILIAS as a library call.

The performance of the ILIAS system can be improved by smarter compilation (e.g., the detection of common subexpressions [13]), and by more efficient parallel algorithms, possibly based on a new distribution.

6.2 Conclusions

The resulting ILIAS system contains an expressive sequential matrix language, which is efficiently implemented on a square torus network of transputers. Key issue in the implementation is avoiding redistributions caused by matrix subscription as much as possible. This is done by using a new data distribution, the grid-base distribution, and by several run-time heuristics.

The feasibility and scalability of the ILIAS system has been demonstrated by timing results for two example ILIAS programs, an LU decomposition and a Strassen matrix multiplication, on transputer networks up to 400 processors. A purely scalar computation is not speeded up by ILIAS. As a general rule, only computations formulated in matrix terms may benefit from a substantial increase of execution speed.

Acknowledgment

The authors wish to thank Rudolf Mak for useful comments on an earlier draft of this paper.
References


In this series appeared:

<table>
<thead>
<tr>
<th>Issue</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/02</td>
<td>R.P. Nederpelt, H.C.M. de Swart</td>
<td>Implication. A survey of the different logical analyses &quot;if...then...&quot;, p. 26.</td>
</tr>
<tr>
<td>91/03</td>
<td>J.P. Katoen, L.A.M. Schoenmakers</td>
<td>Parallel Programs for the Recognition of P-invariant Segments, p. 16.</td>
</tr>
<tr>
<td>91/05</td>
<td>D. de Reus</td>
<td>An Implementation Model for GOOD, p. 18.</td>
</tr>
<tr>
<td>91/06</td>
<td>K.M. van Hee</td>
<td>SPECIFICATIEMETHODEN, een overzicht, p. 20.</td>
</tr>
<tr>
<td>91/07</td>
<td>E. Poll</td>
<td>CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.</td>
</tr>
<tr>
<td>91/10</td>
<td>R.C. Backhouse, P.J. de Bruin</td>
<td>POLYNOMIAL RELATORS, p. 52.</td>
</tr>
<tr>
<td>91/11</td>
<td>R.C. Backhouse, P.J. de Bruin</td>
<td>Relational Catamorphism, p. 31.</td>
</tr>
<tr>
<td></td>
<td>G. Malcolm, E. Voermans, J. van der Woude</td>
<td></td>
</tr>
<tr>
<td>91/12</td>
<td>E. van der Sluis</td>
<td>A parallel local search algorithm for the travelling salesman problem, p. 12.</td>
</tr>
<tr>
<td>91/14</td>
<td>P. Lemmens</td>
<td>The PDB Hypermedia Package. Why and how it was built, p. 63.</td>
</tr>
<tr>
<td>91/16</td>
<td>A.J.J.M. Marcelis</td>
<td>An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.</td>
</tr>
<tr>
<td>Volume</td>
<td>Authors</td>
<td>Titles</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>91/18</td>
<td>Rik van Geldrop</td>
<td>Transformational Query Solving, p. 35.</td>
</tr>
<tr>
<td>91/19</td>
<td>Erik Poll</td>
<td>Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.</td>
</tr>
<tr>
<td>91/23</td>
<td>K.M. van Hee, L.J. Somers, M. Voorhoeve</td>
<td>Z and high level Petri nets, p. 16.</td>
</tr>
<tr>
<td>91/24</td>
<td>A.T.M. Aerts, D. de Reus</td>
<td>Formal semantics for BRM with examples, p. 25.</td>
</tr>
<tr>
<td>91/25</td>
<td>P. Zhou, J. Hooman, R. Kuiper</td>
<td>A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.</td>
</tr>
<tr>
<td>91/27</td>
<td>F. de Boer, C. Palamidessi</td>
<td>Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.</td>
</tr>
<tr>
<td>91/28</td>
<td>F. de Boer</td>
<td>A compositional proof system for dynamic process creation, p. 24.</td>
</tr>
<tr>
<td>91/30</td>
<td>J.C.M. Baeten, F.W. Vaandrager</td>
<td>An Algebra for Process Creation, p. 29.</td>
</tr>
<tr>
<td>91/31</td>
<td>H. ten Eikelder</td>
<td>Some algorithms to decide the equivalence of recursive types, p. 26.</td>
</tr>
<tr>
<td>91/33</td>
<td>W. v.d. Aalst</td>
<td>The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.</td>
</tr>
<tr>
<td>91/34</td>
<td>J. Coenen</td>
<td>Specifying fault tolerant programs in deontic logic, p. 15.</td>
</tr>
</tbody>
</table>
92/01 J. Coenen  
J. Zwiers  
W.-P. de Roever  
A note on compositional refinement, p. 27.

92/02 J. Coenen  
J. Hooman  
A compositional semantics for fault tolerant real-time systems, p. 18.

92/03 J.C.M. Baeten  
J.A. Bergstra  
Real space process algebra, p. 42.

92/04 J.P.H.W.v.d.Eijnde  
Program derivation in acyclic graphs and related problems, p. 90.

92/05 J.P.H.W.v.d.Eijnde  
Conservative fixpoint functions on a graph, p. 25.

92/06 J.C.M. Baeten  
J.A. Bergstra  
Discrete time process algebra, p. 45.

92/07 R.P. Nederpelt  
The fine-structure of lambda calculus, p. 110.

92/08 R.P. Nederpelt  
F. Kamareddine  
On stepwise explicit substitution, p. 30.

92/09 R.C. Backhouse  

92/10 P.M.P. Rambags  
Composition and decomposition in a CPN model, p. 55.

92/11 R.C. Backhouse  
J.S.C.P.v.d.Woude  
Demonic operators and monotype factors, p. 29.

92/12 F. Kamareddine  

92/13 F. Kamareddine  
Set theory and nominalisation, Part II, p. 22.

92/14 J.C.M. Baeten  
The total order assumption, p. 10.

92/15 F. Kamareddine  
A system at the cross-roads of functional and logic programming, p. 36.

92/16 R.R. Seljée  
Integrity checking in deductive databases; an exposition, p. 32.

92/17 W.M.P. van der Aalst  
Interval timed coloured Petri nets and their analysis, p. 20.

92/18 R.Nederpelt  
F. Kamareddine  
A unified approach to Type Theory through a refined lambda-calculus, p. 30.

92/19 J.C.M.Baeten  
J.A.Bergstra  
S.A.Smolka  
Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.

92/20 F.Kamareddine  
Are Types for Natural Language? P. 32.

92/21 F.Kamareddine  
Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
A useful lambda notation, p. 17.
Nominalization, Predication and Type Containment, p. 40.
Bottom-up Abstract Interpretation of Logic Programs, p. 33.
A Programming Logic for Fro, p. 15.
A modelling method using MOVIE and SimCon/ExSpect, p. 15.
A taxonomy of keyword pattern matching algorithms, p. 50.
Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
A continuous version of the Prisoner’s Dilemma, p. 17
QuickSort for linked lists, p. 8.
Deterministic and randomized local search, p. 78.
A congruence theorem for structured operational semantics with predicates, p. 18.
On the unavoidability of metastable behaviour, p. 29
Exercises in Multiprogramming, p. 97
A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
92/22 R. Nederpelt
F. Kamareddine
E. Klein

A useful lambda notation, p. 17.

92/23 F. Kamareddine
M. Codish
D. Dams
Eyal Yardeni

Nominalization, Predication and Type Containment, p. 40.

92/24 M. Codish
D. Dams
Eyal Yardeni

Bottom-up Abstract Interpretation of Logic Programs, p. 33.

92/25 E. Poll

A Programming Logic for Fo, p. 15.

92/26 T. H. W. Beelen
W. J. J. Stut
P. A. C. Verkoulen

A modelling method using MOVIE and SimCon/ExSpect, p. 15.

92/27 B. Watson
G. Zwaan

A taxonomy of keyword pattern matching algorithms, p. 50.

93/01 R. van Geldrop

Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.

93/02 T. Verhoeff

A continuous version of the Prisoner's Dilemma, p. 17

93/03 T. Verhoeff

Quicksort for linked lists, p. 8.

93/04 E. H. L. Aarts
J. H. M. Korst
P. J. Zwietering

Deterministic and randomized local search, p. 78.

93/05 J. C. M. Baeten
C. Verhoef

A congruence theorem for structured operational semantics with predicates, p. 18.

93/06 J. P. Veltkamp

On the unavoidability of metastable behaviour, p. 29

93/07 P. D. Moerland

Exercises in Multiprogramming, p. 97

93/08 J. Verhoosel

A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.

93/09 K. M. van Hee

Systems Engineering: a Formal Approach

93/10 K. M. van Hee

Systems Engineering: a Formal Approach
Part II: Frameworks, p. 44.

93/11 K. M. van Hee

Systems Engineering: a Formal Approach

93/12 K. M. van Hee

Systems Engineering: a Formal Approach
Part IV: Analysis Methods, p. 63.

93/13 K. M. van Hee

Systems Engineering: a Formal Approach

93/14 J. C. M. Baeten
J. A. Bergstra

<table>
<thead>
<tr>
<th>Paper Number</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>93/16</td>
<td>H. Schepers, J. Hooman</td>
<td>A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27</td>
</tr>
<tr>
<td>93/17</td>
<td>D. Alstein, P. van der Stok</td>
<td>Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.</td>
</tr>
<tr>
<td>93/18</td>
<td>C. Verhoef</td>
<td>A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.</td>
</tr>
<tr>
<td>93/19</td>
<td>G-J. Houben</td>
<td>The Design of an Online Help Facility for ExSpect, p.21.</td>
</tr>
<tr>
<td>93/22</td>
<td>E. Poll</td>
<td>A Typechecker for Bijective Pure Type Systems, p. 28.</td>
</tr>
<tr>
<td>93/23</td>
<td>E. de Kogel</td>
<td>Relational Algebra and Equational Proofs, p. 23.</td>
</tr>
<tr>
<td>93/24</td>
<td>E. Poll and Paula Severi</td>
<td>Pure Type Systems with Definitions.</td>
</tr>
<tr>
<td>93/26</td>
<td>W.M.P. van der Aalst</td>
<td>Multi-dimensional Petri nets, p. 25.</td>
</tr>
<tr>
<td>93/27</td>
<td>T. Kloks and D. Kratsch</td>
<td>Finding all minimal separators of a graph, p. 11.</td>
</tr>
<tr>
<td>93/28</td>
<td>F. Kamareddine and R. Nederpelt</td>
<td>A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.</td>
</tr>
<tr>
<td>93/29</td>
<td>R. Post and P. De Bra</td>
<td>GOLD, a Graph Oriented Language for Databases, p. 42.</td>
</tr>
<tr>
<td>93/30</td>
<td>J. Deogun, T. Kloks, D. Kratsch, H. Müller</td>
<td>On Vertex Ranking for Permutation and Other Graphs, p. 11.</td>
</tr>
</tbody>
</table>