# Hierarchical Static Analysis for Improving the Complexity of Linear Algebra Algorithms

Florent de Dinechin, Tanguy Risset and Sophie Robert
IRISA Campus de Beaulieu 35042 Rennes Cedex
e-mail fdupont@irisa.fr, risset@irisa.fr and robert@irisa.fr

This paper presents a methodology for specializing linear algebra programs to improve their complexity. We use MmAlpha, an environment primarily designed for systolic architectures synthesis, to optimize some of the ScaLapack routines for particular matrix structures. These optimizations are based on exact polyhedral computations and demonstrated on the LU factorization algorithm.

## 1. Introduction

Scientific applications provide many time consuming algorithms. Their parallelization is often complex, hence the use of libraries of parallel routines. ScaLapack [2] is the most common of these parallel libraries for linear algebra on dense matrices. It provides solutions to most classical problems, such as for instance the computation of eigenvalues.

In some applications, however, the data have a significant proportion of null values which, if taken into account, reduces the sequential complexity of the algorithms. The particular case we consider here is that of special matrix shapes, such as triangular, tridiagonal, band, etc. Such matrices are not systematically handled optimally by a standard library approach. A solution is to try to specialize the library to such matrices in order to save all the computations involving null values.

This paper describes a general method for this kind of specialization. The method uses MmAlpha, a programming environment for the manipulation of the Alpha language, and is demonstrated on the LU factorization of a square Hessenberg matrix.

## 2. ScaLapack flexibility

ScaLapack derives from the block-partitioned algorithms of Lapack for classical matrices, and from PBlas, a parallelization of the Blas routines. Communications are based on the Blacs kernel which manipulates virtual 2D-grids. Application portability is ensured by the availability of this kernel for several MIMD architectures with distributed memory, and also for workstation networks via PVM or MPI.

On these virtual 2D-grids, the matrices are distributed in a 2D-block-cyclic manner. A descriptor associating the matrix and the informations on its distribution allows a *global* indexing of the matrices which are input to the ScaLapack and PBlas routines. Inside the routines, the descriptor allows the computation of local block indices. For instance, the input parameters of the ScaLapack LU factorization are the size $M \times N$ of the initial matrix $A$, the indices $I_A$, $J_A$ and the descriptor of $A$ in order to compute the factorization of $A' = A(I_A : I_A + M - 1,\ J_A : J_A + N - 1)$.

This allows us to reduce computations in the case of particular matrix shapes by simply modifying the input parameters. For a PBlas routine, these modifications may lead to a call to another PBlas routine, a matrix vector operation becoming a vector vector operation for instance.

## 3. The Alpha language

The figure 1b is an example of an Alpha program, with its sequential version given as the figure 1a. Alpha is a functional language based on the formalism of Affine Recurrence Equations, where variables are data arrays of arbitrary dimensions, and the shape of these data arrays (called their *domain*) may be any finite union of convex polyhedra. Such domains include vectors (in dimension 1), matrices (in dimension 2) but also more specific data structures such as triangular or band or Hessenberg matrices. The domain of a variable is declared in the beginning of the program, see e.g. the two first lines of this program.

There are two classes of operators available to compute on these arrays. Computational *pointwise* operators (`+` and `*` in the figure 1b) describe an operation to be applied to each point of the array. *Spatial* operator describe a transformation of the shape of the array, allowing array aligning and merging. For example, in the figure 1b the `case` operator is a spatial one, merging the data arrays defined by its two sub-expressions.

Since convex polyhedra are finite unions of affine half-spaces, one may show that the set *DOM* of finite unions of convex polyhedra is closed under intersection, union, preimage by an affine function, and image by a certain class of affine functions [7]. This is the basis for the crucial property used in this paper: the operator set in Alpha is defined in such a way that *any expression* of the language has a domain in *DOM* which may be computed *statically.*

For example, as the sum of two arrays is defined where both arrays are defined, the domain of the Alpha expression `exp1+exp2` is the intersection of the domains of `exp1` and `exp2`. In the figure 1b, the domain of the `case`...`esac` expression is the union of the domains of both branches. The domain of the first branch is the intersection of the domain `{i,j| j=1}` and the domain of `A[i,j] * b[j]`, which is computed as previously, the domain of `b[j]` being the preimage of the 1-dimensional domain of `b` by the affine function `i,j->j`. All these computations are implemented in Wilde's *polyhedral library* [7].

This property allows to check automatically that a program verifies the *single assignment rule:* for each point of the domain of each variable, there is one and only one expression defining the value of this variable at this point. This check, and its generalization to structured programs, are performed by the `analyze` tool [3], [5] in MmAlpha, an environment for the manipulation of Alpha programs.

## 4. Principle of program specialization

The purpose of this section is to introduce on a simple example the principle and the technique used for specialization of ScaLapack routines. This example consists in a matrix

```
                          b :   {i|1 ≤ i ≤ N} of real;
                          A : {i,j|1 ≤ i ≤ N; 1 ≤ j ≤ N}  of real;
    Do 10 i = 1, N        temp :   {i,j|1 ≤ i ≤ N; 1 ≤ j ≤ N} of real;
      temp = A[i,1]*b[1]  let
      Do 20 j=2,N           temp[i,j] =
        temp = temp +         case
          A[i,j]*b[j]         {i,j|j = 1} :  A[i,j] * b[j];
20    End Do                  {i,j|2 ≤ j} :  temp[i,j-1] + A[i,j] * b[j];
      u[i]=temp               esac;
10  End Do                tel;

     (a)                                (b)
```

Figure 1. The product of a $N \times N$ matrix $A$ by a vector $b$. (a) imperative code (b) ALPHA program

vector product $Hb$ when the matrix $H$ is an Hessenberg matrix (an Hessenberg matrix is upper triangular with one sub-diagonal). Operating on such matrices using algorithm for regular matrices will execute useless operations, because almost half of the coefficients are null. The use of the MMALPHA environment and of the polyhedral library will help in specializing the code for Hessenberg matrices by propagating 0 values. The proposed method is divided in 5 steps:
1) translate the imperative code into ALPHA code;
2) use MMALPHA tools to find out places where null data is used;
3) transform the ALPHA specification in order to remove completely all use of null value in computations;
4) deduce the reduction of the domains of the data used in computations;
5) translate back the specification into imperative code.

Starting from the ALPHA specification described in figure 1, if we *artificially* restrict the input matrix `A` to be present on the domain $\{i, j \mid 1 \le i, j \le N; i \le j + 1\}$ (the domain on which a Hessenberg matrix is not null), the single assignment rule will no more be respected. We can use the `analyze` command of MMALPHA to find out which variable of the program are altered, that is, on which variable definition a null value has an influence. Here, it is obviously on `temp`. Therefore, for each case branch in the definition of `temp`, we analyze recursively the tree of the expression, starting from the leaves. If a subexpression contains the `A` variable, we compute its new domain which is in fact, because of the artificial restriction of the domain of `A`, the domain where the expression does not use a null value of `A`. This allows us to specialize the expression to this domain.

Here `A` appears for example in a multiplication in the second case branch of the definition of `temp`. We compute the domain of this multiplication `d=expDomain["A[i,j]*b[j]"]`. This MMALPHA command returns `d` $= \{i, j | j + 2 \le i \le N; 1 \le j\}$. Then we split this second case branch in two: one sub-branch where the expression is unchanged, because it involves non null values of `A`, and one where we replace `A` with 0 (because 0 is absorbent for the multiplication).

The domain of the first sub-branch is the intersection of `d` and the domain of the initial case branch: `d1 = DomIntersection[d,` $\{i, j|2 \leq j\}$`]`. The domain of the second sub-branch is the complementary of `d` in the initial case branch, computed as
`d2 = DomDifference[ expDomain["`$\{i, j|2 \leq j\}$`:A[i,j]*b[j]"], d ];`

After this case splitting transformation has been applied to both case branches of program of figure 1, we obtain:

```
{i,j| 1≤ i≤ 2; j=1} :  A[i,j] * b[j];
{i,j | 3≤ i≤ N; j=1}:  0[];
{i,j | 1≤ i≤ (j+1,N); 1<j≤ N} :  temp[i,j-1] + A[i,j]*b[j];
{i,j | j+2≤ i≤ N; 1<j} :  temp[i,j-1];
```

This program now respects the single assignment rule, as the `analyze` tool may check.

We need now to propagate the zero values in the program. In our case, it seems quite obvious that the `temp` variable will be null on the domain $d' = \{i, j|j+2 \leq i \leq N; 1 \leq j\}$. Indeed, the `0` value is initialized by the second case branch and this value is transmitted everywhere on this domain $d'$ by the last case branch. However, such property may be difficult to prove when more complex domains are involved. The MmAlpha tools can help in guessing what is the right hypothesis and in proving it. The demonstration of this property for the `temp` variable is done with an inductive reasoning (similar to the one used in [4]). This technique is beyond the scope of this paper. Here, after applying this reasoning we can replace the last three case branches by :

```
{i,j | 1≤ i≤ (j+1,N); 1<j≤ N} :  temp[i,j-1] + A[i,j]*b[j];
{i,j | j+2≤ i≤ N; 1≤ j} :  0[];
```

The next step consists in removing physically the useless case branches with `0` values. The simple use of the MmAlpha function `cut` allows to substitute the `temp` variable by two new ones: `temp1`, where `temp` was null, and `temp2` where it wasn't. After this treatment, we obtain the following declarations:

```
temp1 :  {i,j | j+2≤ i≤ N; 1≤ j} of real;
temp2 :  {i,j | 1≤ i≤ (N,j+1); 1≤ j≤ N}
```

Then the code is this one where the `temp` is replaced by `temp1` and `temp2` (note that `temp1` is now useless) :

```
temp1[i,j] =        0[];
temp2[i,j] =    case
                {i,j| 1≤ i≤ 2; j=1} :  A[i,j] * b[j];
                {i,j| i=j+1; 2≤ j≤ N-1} :  A[i,j] * b[j];
                {i,j| 1≤ i≤ j; 2≤ j≤ N} :  temp2[i,j-1] + A[i,j] * b[j];
                esac;
u[j] =  temp2[j,N];
```

This illustrates also the step four of the method: reduction of the domains of the computations. Here the variable of `temp` has been replaced with `temp2` which has a smaller domain.

Finally, we can easily translate back this new specification in imperative code (it is very close to the original specification, only bounds of domains changed), and one can check that the number of multiplication-addition for performing the matrix vector multiplication was decreased from $N^2$ to $\frac{N^2+3N-2}{2}$.

## 5. Application to the ScaLapack LU factorization

This section is devoted to the use of the previous description on a representative example: the block partitioned ScaLapack algorithm of the LU factorization [1]. The aim is to specialize the corresponding routine to Hessenberg matrices. In order to keep the bidiagonal shape of the $L$ matrix of this factorization, we consider a version without partial pivoting.

Suppose that $(L_{i1})_{1\leq i\leq N}$ and $(U_{1j})_{1\leq j\leq N}$ have already been computed. It remains to compute the matrix blocks in bold characters in the equation (1).

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & \mathbf{L_{22}} & 0 & 0 \\ L_{31} & \mathbf{L_{32}} & I & 0 \\ L_{41} & \mathbf{L_{42}} & 0 & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & \mathbf{U_{22}} & \mathbf{U_{23}} & \mathbf{U_{24}} \\ 0 & 0 & \mathbf{\tilde{A}_{33}} & \mathbf{\tilde{A}_{34}} \\ 0 & 0 & \mathbf{\tilde{A}_{43}} & \mathbf{\tilde{A}_{44}} \end{pmatrix}. \tag{1}$$

The operations which compose the main routine PDGETRF are the following.

- PDGETF2: $\begin{pmatrix} \mathbf{L_{22}} \\ \mathbf{L_{32}} \\ \mathbf{L_{42}} \end{pmatrix}$ is calculated from a Gaussian elimination of $\begin{pmatrix} A_{22} \\ A_{32} \\ A_{42} \end{pmatrix}$.
- PDTRSM: $L_{22}(\mathbf{U_{22}}\mathbf{U_{23}}\mathbf{U_{24}}) = (A_{22}A_{23}A_{24})$.
- PDGEMM: $\begin{pmatrix} \mathbf{\tilde{A}_{33}} & \mathbf{\tilde{A}_{34}} \\ \mathbf{\tilde{A}_{43}} & \mathbf{\tilde{A}_{44}} \end{pmatrix} = \begin{pmatrix} A_{33} & A_{34} \\ A_{43} & A_{44} \end{pmatrix} - \begin{pmatrix} L_{32} \\ L_{42} \end{pmatrix} (U_{22}U_{23})$.

We start with an Alpha specification following the ScaLapack structure. Each subsystem represents either a ScaLapack routine either a PBlas routine. In our Alpha description we only take the computations into consideration (and not the communications) and by this way we keep a global indexing. The modifications of the specification due to the transformations as described in the section 4 will allow us to adapt the input parameters of the ScaLapack routines involved.

### 5.1. The initial specification

First we have to choose an Alpha variable to represent the block partitioned matrices. In the ScaLapack, the representation uses the size of the matrix $M \times N$ and the block size $M_B \times N_B$. In our example $M = N$ and $M_B = N_B$. The Alpha variable corresponding to the matrix is declared as:

`A`:{i,j,p,q | 1≤i,j≤ $N_B$; 1≤p,q≤ $R_B$} `of real;`

where $R_B$ is the number of row or column of blocks ($N = N_B \times R_B$).
From these initial declarations we have to define a specification of each function involved in the LU factorization. This specification computes a variable LU declared by:

`LU` : {i,j,p,q,k | 1≤i,j≤ $N_B$; 1≤p,q≤ $R_B$; 0≤k< $R_B$} `of real;`

The index $k$ is the number of the step such that all the blocks $(L_{ij})_{1\leq j<k}^{j\leq i\leq R_B}$ and $(U_{ij})_{i\leq j\leq R_B}^{1\leq i\leq k}$ are already computed and such that $(L_{ik})_{k\leq i\leq R_B}$, $(U_{kj})_{k\leq j\leq R_B}$ and $(\tilde{A}_{ij})_{k+1\leq i,j\leq R_B}$ are going to be defined as explained in (1).
Let us simplify the explanation in considering only the operation realized by PDGEMM. At the step $k$ the call in the routine PDGETRF is :

```
CALL PDGEMM('N','N',N-J-NB+1,N-J-NB+1,NB,-ONE,A,J+NB,J,DESCA,A,
                  J,J+NB,DESCA,ONE,A,J+NB,J+NB,DESCA).
```

It realizes the operation described by the figure 2(a) (for $k = 2$ thus $J = N_B + 1$). In its translation to ALPHA, the figure 2 shows the correspondence between the ALPHA routine PDGEMM$_{alpha}$ inputs variables $A_1$, $B_1$ and output variable $C_1$ and the A matrix (it is to say the LU matrix at the step $k = 2$). The ScaLapack indexing from which we deduce these input and output variables is also indicated.

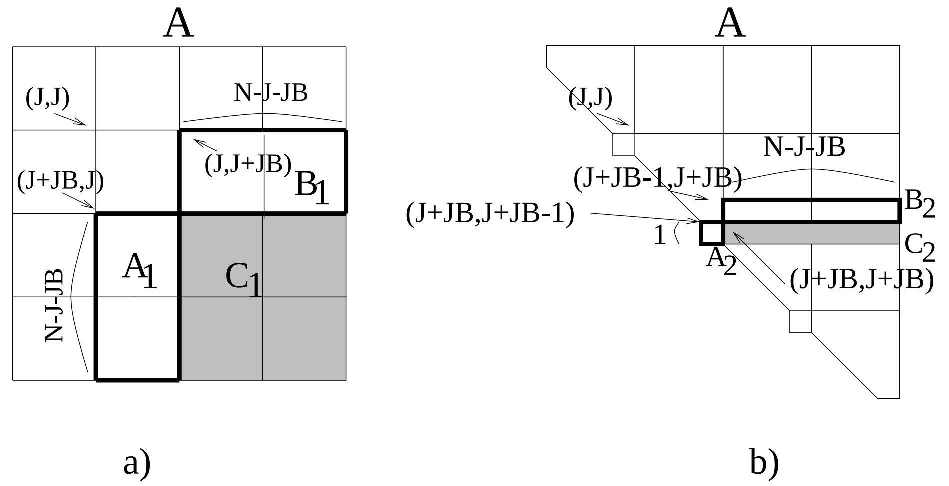


Figure 2. The computation realized on the matrix $A$ by the call to PDGEMM for $k = 2$ in the original routine (a) and after the specialization (b). In the ALPHA specialization, bold blocks represent the inputs of the routine and gray blocks the computation executed by this routine.

### 5.2. The LU specialization

As described in section 4, the initial matrix $A$ is replaced by a new declaration $H$ according to the Hessenberg shape of interest which is declared as:

H : {i,j,p,q | p=q; 1≤q≤ $R_B$; 1≤i≤(j+1,$N_B$); 1≤j≤ $N_B$} |
{i,j,p,q | p=q+1; 1≤q< $R_B$; i=1; j=$N_B$} |
{i,j,p,q | 1≤p<q; 1≤q≤ $R_B$; 1≤i,j≤ $N_B$} of real;

where | denotes domains union. Then we prove that the matrix LU is null except on the domain :

$$\{i,j,p,q,k \mid i=1;\ j=N_B;\ 1\leq p \leq(R_B,q+1);\ 1\leq q\leq R_B;\ 0\leq k\leq R_B\} \mid$$
$$\{i,j,p,q,k \mid 1\leq i\leq(N_B,j+1);\ 1\leq j\leq N_B;\ 1\leq p\leq q;\ q\leq R_B;\ 0\leq k\leq R_B\} \mid$$
$$\{i,j,p,q,k \mid 1\leq i\leq N_B;\ 1\leq j\leq N_B;\ 1\leq p\leq q-1;\ q\leq R_B;\ 0\leq k\leq R_B\}$$

This demonstration can be done using a generalization of the technique mentioned in the section 4 to structured systems.
The new declarations of the input variables and of the LU variable lead to some errors detected by the `analyze` command. For example the variable $A_1$ has been altered by specializations prior to those of PDGEMM$_{alpha}$. For the call to the system PDGEMM$_{alpha}$ the `analyze` command indicates the error:

```
ERROR: In call pdgemm, input A1 not assigned over the domain :
{i,j,p,k | 2<=i<=NB; 1<=j<= NB; k+1<=p<= RB; 1<=k} |
{i,j,p,k | 1<=i<=NB; 1<=j<=NB-1; k+1<=p<=RB; 1<=k} |
{i,j,p,k | 1<=i<=NB; 1<=j<=NB; k+2<=p<=RB; 1<=k}
```

This error means that the data may not be used on this domain (it is null). So this leads us to define in the subsystem PDGEMM$_{alpha}$ the correct domain of the corresponding input variable (a difference between the initial domain and those indicated by the error). From this modification and the scheme described in the section 4 we can proceed to its specialization for all $k$.
From the `analyze` indications, each input variable domain is corrected allowing us to specialize the code of the systems. Hence a new Alpha specification adapted to the Hessenberg shape of the initial matrix.
The last step is to translate back this specification in term of global indexing of the ScaLapack routines. For example let us consider again the system PDGEMM$_{alpha}$. The figure 2(b) describes its specialization. The new Alpha variables ($A_2$, $B_2$ and $C_2$ in grey) allow to define the new global indexing and so the new call in the routine PDGETRF:

```
CALL PDGEMM('N','N',1,N-J-JB+1,1,-ONE,A,J+JB,J+JB-1,DESCA,
     A,J+JB-1,J+JB,DESCA,ONE,A,J+JB,J+JB,DESCA)
```

In this case the matrix operation could be reduced to a vectorial one.

## 6. Conclusion

We have proposed a methodology for the specialization of ScaLapack routines with the MmAlpha environment. This methodology allowed us to derive a LU factorization function with a complexity of $O(N^2)$ from the initial $O(N^3)$ PDGETRF function. However parallelization overhead and communications may greatly influence the computation time, therefore the new routine has been tested on an Intel-Paragon multiprocessor. The figure 3 gives the gain between this implementation and the initial routine. It shows that our specialization is really useful, especially if the number of processors available is small. This technique can be used for providing specialized versions of other classical ScaLapack functions for Hessenberg matrices, but also for developing such functions for other kind of matrices (triangular, bidiagonal or tridiagonal), if they do not exist. At this point, however, our technique cannot modify communications routines, partly because of the restrictions of the MmAlpha tool which can only handle convex polyhedra.

| processor grid type | $N_B = 10$ | $N_B = 100$ |
|---|---|---|
| $2 \times 2$ | 76 % | 77 % |
| $3 \times 3$ | 62 % | 70 % |
| $4 \times 5$ | 38 % | 32 % |

Figure 3. Acceleration on a Intel-Paragon multiprocessor of the PDGERTF routine after specialization.

## REFERENCES


1. J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. Whaley. The design and implementation of the ScaLapack LU, QR, and Cholesky factorization routines. Technical report, Oak Ridge National Laboratory, 1994.
2. J. Choi, J. Dongarra, R. Pozo, and D. Walker. A scalable linear algebra library for distributed memory concurrent computers. In IEEE Computer Society Press, editor, *Proceedings of Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127, Los Alamitos, California, 1992.
3. Florent de Dinechin and Sophie Robert. Hierarchical static analysis of structured systems of affine recurrence equations. In *Application Specific Array Processors*. IEEE Computer Society Press, August 1996.
4. C. Dezan and P. Quinton. Verification of regular architectures using Alpha : a case study. In *ASAP'94*, pages 164–176, San Francisco, August 1994. IEEE Computer Society Press.
5. Florent Dupont de Dinechin. *Systèmes structurés d'équations récurrentes : mise en œuvre dans le langage Alpha et applications*. Thèse de doctorat, université de Rennes I, January 1997.
6. K.C Toh and L.N. Trefethen. Calculation of pseudospectra by the Arnoldi iteration. *SIAM J. Sci. Comp.*, 17:1–15, 1996.
7. D. Wilde. A library for doing polyhedral operations. Publication Interne 785, IRISA, Rennes, France, December 1993. Also published as INRIA Research Report 2157.