<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>A Parallel Algorithm for the Solution of the Deconvolution Problem in Heterogeneous Networks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Authors(s)</strong></td>
<td>Alonso, Pedro, Lastovetsky, Alexey, Vidal, Antonio M.</td>
</tr>
<tr>
<td><strong>Publication date</strong></td>
<td>2005-12-19</td>
</tr>
<tr>
<td><strong>Series</strong></td>
<td>UCD CSI Technical Reports, ucd-csi-2006-2</td>
</tr>
<tr>
<td><strong>Publisher</strong></td>
<td>University College Dublin. School of Computer Science and Informatics</td>
</tr>
<tr>
<td><strong>Item record/more Information</strong></td>
<td><a href="http://hdl.handle.net/10197/12354">http://hdl.handle.net/10197/12354</a></td>
</tr>
</tbody>
</table>

The UCD community has made this article openly available. Please share how this access benefits you. Your story matters! (@ucd_oa)
A Parallel Algorithm for the Solution of the Deconvolution Problem in Heterogeneous Networks*†

Pedro Alonso, Alexey Lastovetsky, Antonio M. Vidal

University College Dublin, School of Computer Science and Informatics
December 19, 2005

Abstract

In this work we present two parallel algorithms for the solution of a given least squares problem with structured matrices. This problem arises in many applications most related to digital signal processing, an example is given. Both parallel algorithms have been designed to speed-up the sequential one in a heterogeneous network of computers. They differ from the approximation followed to implement parallel algorithms on heterogeneous networks of computers known as HeHo and HoHe strategies. However, our study goes beyond the practical usefulness of our heterogeneous parallel application. One one hand, the results obtained validates the recent developed HeteroMPI as a very useful tool for programming heterogeneous parallel algorithms. On the other hand, although HeteroMPI has initially been designed to apply the HeHo strategy, we propose a way this tool can be used in the HoHe strategy. Pros and cons of the use of HeteroMPI for both strategies will be deeply study through the application example.

*Funded by Secretaría de Estado de Universidades e Investigación del Ministerio de Educación y Cultura of Spain.
†Supported by Spanish MCYT and FEDER under Grant TIC 2003-08238-C02-02.
Contents

1 Introduction 2

2 The Multichannel Sound Reproduction System 4

3 The Sequential Algorithm 6
   3.1 Problem Solution ........................................ 6
   3.2 Block–Toeplitz matrices .................................. 7
   3.3 Rank Displacement and Block–Toeplitz matrices ........ 7
   3.4 Triangular decomposition of symmetric Cauchy–like matrices . 9
   3.5 Implementation of the sequential algorithm .......... 12

4 The Parallel Algorithm, the HeHo strategy 15
   4.1 The Parallel Algorithm .................................... 16
   4.2 Parallel triangularization of symmetric Cauchy–like matrices . 16
   4.3 The Performance Model .................................... 21
   4.4 Implementation of the Parallel Algorithm .......... 25
   4.5 Experimental Results .................................... 28

5 The Parallel Algorithm, the HoHe strategy over HeteroMPI 29
   5.1 Data distribution ........................................ 30
   5.2 Modified ScaLAPACK Tools .............................. 32
   5.3 Implementation of the Parallel Algorithm .......... 36

6 Conclusions 39

A Code of HeteroScaLAPACK routines 42

1 Introduction

The $mpC$ is a programming language for writing parallel programs for Heterogeneous Networks of Computers (HNoC’s) [1]. $mpC$ is an extension to the C[] language, which is a Fortran 90 like extension to ANSI C supporting array–based computations. $mpC$ provides the programmer a useful tool to describe the performance model of the parallel algorithm. The programming system uses this information to optimally map at runtime this algorithm to the computers of the executing network. The $mpC$ programming system employs an advanced performance model of a heterogeneous NoC’s. As a result, $mpC$ allows the programmer to write an efficient program for heterogeneous networks in a portable form. The program will be automatically tuned at runtime to each executing NoC’s trying to run on the network with maximal possible speed.

Heterogeneous Message Passing Interface (HeteroMPI) is an extension of MPI obtained by applying the $mpC$ parallel programming model to the message–passing library [2]. Actually, HeteroMPI is an adaptation of $mpC$ language to the MPI programming level. The main idea of HeteroMPI is to automate the
1 INTRODUCTION

selection of groups of processes that will be executed together in the same physical processor. The number of such number of processes is runtime tuned by the mpC programming environment due to HeteroMPI uses essentially the same performance model of mpC. Thus, HeteroMPI allows the application programmers to describe a performance model of their homogeneous algorithm to be run in a heterogeneous NoC’s thanks to the HeteroMPI interface.

At present, there exists a HeteroMPI version available to be used [3]. The aim of this paper is to present an example of use of HeteroMPI to develop a heterogeneous parallel algorithm based on a homogeneous one. The key of the work falls into the accuracy of the performance model of the homogeneous parallel algorithm. Once it has been build, other important arguments that affect the performance of the execution like the block size or the number of processes has to been correctly chosen to obtain the maximum possible performance whatever the heterogeneous network is used.

The application on which is going to work deals with a particular issue of digital signal processing called inverse filtering of multichannel systems. However, the solution of underlying mathematical problem covers a more wider field of related digital signal analysis problems. Furthermore, the acquired experience can be generalised to the expected results of a larger set of homogeneous parallel triangularization algorithms on heterogeneous networks using HeteroMPI.

Inverse filtering and equalization of multichannel systems is a field of growing interest. This fact is mainly due to the upcoming applications of multichannel systems such as digital communication (mainly new generation digital mobile communications that incorporates array processing at the base stations) and the modern multichannel audio reproduction systems such as three-dimensional (3-D) audio [4], or active noise control, and the availability of new technology resources which make possible the implementation of more complex signal processing algorithms.

The mathematical model of inverse filtering and equalization multichannel systems are standing for large-scale matrix problems with structure. The major challenge in this area is to design fast and numerically reliable algorithms for large-scale structured linear matrix equations and the least squares matrix problem. For small-size problems, there is often not much else to do except to use one of the already standard methods of solution such as Gaussian elimination or the QR decomposition. However, as the problem size increases it is important to identify special structures in order to reduce the computational burden. A very extensive work have been made in this way and, as a result of, many algorithms have been developed exploiting the special structure.

Several algorithms have been traditionally used to solve systems of equations or the linear least squares problem of Toeplitz-like matrices exploiting its special structure to get a computational cost an order of magnitude lower than other classical algorithms for non-structured matrices. These are the well-known fast algorithms. For example, the Levinson–Durbin recursion has been proved to be very useful in several cases [5]. A generalized block processing version of the Levinson–Durbin algorithm [6] was used in [7] and [8], to design inverse filters for cross-talk cancellation. Other algorithm can be found in [9].
To develop the heterogeneous parallel algorithm we start from a homogeneous parallel algorithm for the solution of the inverse filtering multichannel systems presented in [10] but modified to use a more efficient approach based on the concept of Cauchy-like matrix. The use of Cauchy-like matrices to design algorithms for the solution of standard numerical linear algebra problems like the linear systems solution or the minimization of the least squares problem with structured matrices has been successfully applied in complex, hermitian, real and symmetric real cases [11, 12, 13, 14, 15]. A brief description of the application example is done in Section 2. The mathematical and algorithmic background is explained with detail in Section 3. The homogeneous parallel algorithm with the modifications needed to run on a heterogeneous NoC’s is presented in Section 4. This approximation is known as the HeHo strategy as it will explained there. However, we have go further on by developing the same parallel algorithm with a heterogeneous distribution of data. This is a new idea that, still based on the HeteroMPI tool, allows the programmer to use a HoHe strategy as it will be described in Section 5. The work finishes with a conclusions section with open issues to be affordable in the future.

2 The Multichannel Sound Reproduction System

Fig. 1 shows a block diagram of a multichannel sound reproduction system. The configuration showed in Fig. 1 is typical in multichannel sound reproduction systems, where inverse filters are usually calculated using the least squares method in the time domain [16]. The reproduction system renders \( K \) input signals into the listening space. A block labeled \( H \) filters these input signals prior to feeding the electroacoustic system. As illustrated in Fig. 1, each input signal is filtered through an array of filters whose impulse responses are denoted by \( h_{i,j}[n] \), where \( i \) indicates which source is reproducing the filter output and \( j \) which input signal is filtered by this filter. A different set of filters is chosen depending on the desired application. Cross-talk cancellation, inverse filtering, equalization and virtual source positioning represent some examples of these applications. Regardless of the selection of the desired application, physical and computational boundaries will constrain the filter design. Therefore, efficient and practical computational methods are needed to carry out this design task.

The inverse filtering problem in practical multichannel audio reproduction systems basically consists of building a matrix of digital finite duration filters (a different filter vector for each signal to be rendered), whose convolutions with the signal transmission channels best approximate a desired response.

The filter matrix is calculated using the electroacoustic system (or transmission channels) actual responses, which were previously measured, and the desired signals at the listening or reproduction points. Electroacoustic system responses are also usually modeled as finite duration filters. From a mathematical point of view, the multichannel inverse filtering problem is usually resolved.
using a linear set of equations that results from the linear convolutions of the filter matrix and the electroacoustic system responses. However, it may not be possible in practice to obtain a squared linear set of equations, hence a least squared error solution is preferred.

The electroacoustic system responses will be modeled as FIR filters of \( n_c \) coefficients, and \( c_{i,j}(k) \) will be the \((k+1)\)th coefficient of the channel between the \( j \)th source (loudspeaker) and the \( i \)th listening point (microphone). Every inverse filter will have \( n_h \) coefficients and be denoted by \( h_{i,j} = [h_{i,j}(0), \ldots, h_{i,j}(n_h - 1)]^T \); this filter processes the \( j \)th input signal to feed the \( i \)th loudspeaker. By applying the superposition principle, each input signal can be separately considered. Thus, each input signal has its corresponding desired signal at every listening point. These desired signals will be vectors of \((n_c + n_h - 1)\) coefficients, \( a_{i,j} = [a_{i,j}(0), \ldots, a_{i,j}(n_c + n_h - 2)]^T \), where subscript \( i \) corresponds to the listening point and subscript \( j \) corresponds to the input signal. Each input signal is filtered through a different set of filters. Therefore, the set of filters that processes a given input signal can be independently calculated without any loss of generality. A single input signal is usually assumed throughout this paper.

A reproduction system with \( L \) loudspeakers and \( M \) microphones can be expressed as

\[
[e]_{(n_c+n_h-1)M \times 1} = [C]_{(n_c+n_h-1)M \times n_h L} [h]_{n_h L \times 1} - [a]_{(n_c+n_h-1)M \times 1}, \tag{1}
\]

where \( h^T = [h_{1,1}^T, \ldots, h_{L,1}^T] \), \( a^T = [a_{1,1}^T, \ldots, a_{M,1}^T] \), \( e^T = [e_{1,1}^T, \ldots, e_{M,1}^T] \), and

\[
[e]_{(n_c+n_h-1)M \times 1} = [C]_{(n_c+n_h-1)M \times n_h L} [h]_{n_h L \times 1} - [a]_{(n_c+n_h-1)M \times 1}, \tag{1}
\]
3 THE SEQUENTIAL ALGORITHM

Matrix $C$ is composed by $M \times L$ blocks $C_{i,j}$ of the form

$$C_{i,j} = \begin{bmatrix}
c_{i,j}(0) & 0 & 0 \\
c_{i,j}(1) & c_{i,j}(0) & \ddots \\
\vdots & c_{i,j}(1) & \ddots \\
c_{i,j}(n_c - 1) & \vdots & c_{i,j}(0) \\
0 & c_{i,j}(n_c - 1) & c_{i,j}(1) \\
\vdots & \ddots & \vdots \\
0 & \cdots & c_{i,j}(n_c - 1)
\end{bmatrix}.$$  \hspace{1cm} (2)

Expression (1) represents a system of equations where matrix $C$ is a non-square matrix composed by nonsquare Toeplitz blocks. A least squares error criteria is commonly used to calculate the array of inverse filters. Thus, the array of inverse filters in (1), $h$, is chosen to minimize the squared error,

$$\min_h \{\|e\|_2^2\} = \min_h \{\|Ch - a\|_2^2\}.$$  \hspace{1cm} (3)

3 The Sequential Algorithm

The problem exposed in Section 2 is only an example of use of the mathematical solution that is proposed in this section to be solve in a Heterogeneous NoC’s. Through the following subsections we explain the mathematical problem and its solution by means of a sequential algorithm first to expose the parallel solution on homogeneous platforms as on the heterogeneous ones.

3.1 Problem Solution

The solution of a general least squares problem (as the one proposed in (3)) defined as

$$\min_x \|Mx - b\|_2^2,$$  \hspace{1cm} (4)

where $M$ is a structured matrix like a Toeplitz matrix, a product of Toeplitz matrices or a lot of more examples, can be approached by means of the solution of the associated normal equations

$$M^T M x = M^T b.$$  

The solution of the normal equations can be obtained triangularizing the product $M^T M = LDL^T$, where $L$ is a unit lower triangular matrix and $D$ is diagonal. Numerically, this method is not recommended for general non-structured matrices because there exist a lost of accuracy if the product $M^T M$ is explicitly done. However, this is not the case of structured matrices because this product is not explicitly carried on as it is further explained. Thus, in order to solve a general least squares problem (4) the algorithms obtain the semi-normal equations

$$LDL^T x = M^T b,$$
from the normal equations to solve the triangular linear systems that lead to the solution \( x \) of (4).

### 3.2 Block–Toeplitz matrices

The structured matrix we work with is a Toeplitz–block one. It is not hard to show that, when matrix \( M \) (4) has the form of the matrix \( C \) defined in (1) and (2), the product \( T = M^T M \) is Toeplitz–block, that is,

\[
T = \begin{pmatrix}
T_{0,0} & T_{0,1} & \cdots & T_{0,\nu-1} \\
T_{1,0} & T_{1,1} & \ & \\
& \ddots & \ddots & \\
T_{\nu-1,0} & \cdots & T_{\nu-1,\nu-1}
\end{pmatrix},
\]

where each \( T_{ij} \in \mathbb{R}^{n/\nu \times n/\nu} \), for \( i, j = 0, \ldots, \nu - 1 \), is a scalar Toeplitz matrix and \( \hat{b}, \hat{x} \in \mathbb{R}^n \) are the independent and the unknown vectors, respectively.

There exist another type of structured matrices called block–Toeplitz matrix and defined as

\[
\hat{T} = \begin{pmatrix}
B_0 & B_{-1} & \cdots & B_{1-n/\nu} \\
B_{1} & B_0 & \ & \\
& \ddots & \ddots & \\
B_{n/\nu-1} & \cdots & B_0
\end{pmatrix},
\]

with each \( B_i \in \mathbb{R}^{n/\nu \times n/\nu} \), for \( i = 1 - n/\nu, \ldots, n/\nu - 1 \), being a non–structured matrix and with \( b, x \in \mathbb{R}^n \) being the independent and the unknown vectors, respectively.

Matrices (6) and (5) are related by means of a permutation matrix \( P \) so that \( T = P \hat{T} P^T \). This fact makes possible to build block algorithms to solve systems with the form (5) by means of the solution of a system of the form (6) exploiting the use of BLACS3 operations. However, as it has been shown in [17], more performance can be achieved by using parallel algorithms that work directly on Toeplitz–block matrices. Furthermore, the Cauchy–like matrix approach, as it is use in the present paper, offers more performance by using this last type of algorithms.

### 3.3 Rank Displacement and Block–Toeplitz matrices

It is said that a matrix of order \( n \) is structured if its displacement representation has a lower rank regarding \( n \). The displacement representation of a symmetric Toeplitz matrix \( T \in \mathbb{R}^{n \times n} \),

\[
T = \begin{pmatrix}
t_0 & t_1 & t_2 & \cdots & t_{n-1} \\
t_1 & t_0 & t_1 & \cdots & t_{n-2} \\
t_2 & t_1 & t_0 & \cdots & \\
\vdots & \ddots & \ddots & \ddots & \\
t_{n-1} & t_{n-2} & t_1 & t_0
\end{pmatrix},
\]
can be defined in several ways depending on the form of the displacement matrices. A useful form for our purposes is

$$\nabla_F T = F^T T F = \mathcal{G} \mathcal{H} \mathcal{G}^T ;$$  \hspace{1cm} (7)

where

$$F = \begin{pmatrix}
0 & 1 & 0 & \ldots & 0 \\
1 & 0 & 1 & \vdots & \\
0 & 1 & 0 & \ddots & 0 \\
\vdots & \ddots & \ddots & \ddots & 1 \\
0 & \ldots & 0 & 1 & 0
\end{pmatrix},$$

called displacement matrix, $\mathcal{G} \in \mathbb{R}^{n \times 4}$ is the generator matrix and $\mathcal{H} \in \mathbb{R}^{4 \times 4}$ is a skew-symmetric signature matrix. The rank of $\nabla_F T$ is 4, that is, lower than $n$ and independent of $n$.

It is easy to see that the displacement of $T$ with respect to $F$ is a matrix of a considerably sparsity from which it is not difficult to obtain an analytical form of $\mathcal{G}$ and $\mathcal{H}$.

A symmetric Cauchy-like matrix $C$ is a structured matrix that can be defined as the unique solution of the displacement equation

$$\nabla_A C = \Lambda C - C \Lambda = \hat{\mathcal{G}} \mathcal{H} \hat{\mathcal{G}}^T ,$$  \hspace{1cm} (8)

being $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$, where $\text{rank}(\nabla_A C) \ll n$ and independent of $n$.

We define the unnormalized Discrete Sine Transformation (DST) as

$$\hat{S} = \sin \frac{i j \pi}{n+1} ; \quad i, j = 1, \ldots, n,$$  \hspace{1cm} (9)

and the normalized DST as

$$S = \sqrt{\frac{2}{n+1}} \hat{S} .$$  \hspace{1cm} (10)

Matrix $S$ is symmetric and orthogonal and matrix $F$ can be diagonalized by means of the DST $S$, that is, $SF S = \Lambda$ where

$$\Lambda = \text{diag} \left( 2 \cos \frac{j \pi}{n+1} \right) , \quad j = 1, \ldots, n .$$

Now, we use the normalized Discrete Sine Transformation (DST) $S$ as defined in [18]. Since $S$ is symmetric, orthogonal and $SFS = \Lambda$ [19, 20], we obtain

$$S(FT - TF)S = S(\mathcal{G} \mathcal{H} \mathcal{G}^T)S \quad \Rightarrow \quad \Lambda C - CA = \hat{\mathcal{G}} \mathcal{H} \hat{\mathcal{G}}^T ,$$

where $C = STS$ and $\hat{\mathcal{G}} = S \mathcal{G}$. This shows how it can be transformed (7) into (8).
The displacement property can be easily generalized to the Toeplitz–block matrix $\mathbf{T}$ (5),
\[ \nabla_F \mathbf{T} = \mathbf{F} \mathbf{T} - \mathbf{T} \mathbf{F}, \tag{11} \]
where
\[ F = \begin{pmatrix} F & \cdots & F \\ F & \ddots & \vdots \\ \vdots & \ddots & F \end{pmatrix} \]
Now we build the following transformation
\[ S = \begin{pmatrix} S & \cdots & S \\ S & \ddots & \vdots \\ \vdots & \ddots & S \end{pmatrix} \]
to be applied to the displacement representation (11) giving
\[ S(\mathbf{F} \mathbf{T} - \mathbf{T} \mathbf{F})S = (S \mathbf{F} S)(S \mathbf{T} S) - (S \mathbf{T} S)(S \mathbf{F} S) = \tilde{\Lambda} \mathbf{C} - \mathbf{C} \tilde{\Lambda}, \tag{12} \]
being
\[ \tilde{\Lambda} = \begin{pmatrix} \Lambda & \cdots & \cdots & \cdots \\ \Lambda & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \cdots & \ddots & \ddots \end{pmatrix}, \]
and
\[ \tilde{\mathbf{C}} = \begin{pmatrix} S \mathbf{T}_{0,0} S & S \mathbf{T}_{0,1} S & \cdots & S \mathbf{T}_{0,\nu-1} S \\ S \mathbf{T}_{1,0} S & S \mathbf{T}_{1,1} S & \cdots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ S \mathbf{T}_{\nu-1,0} S & \cdots & S \mathbf{T}_{\nu-1,\nu-1} S \end{pmatrix}, \]
Displacement representations (11) and (12) have a rank of $4\nu$ so the generators have $4\nu$ columns
\[ \nabla_F \mathbf{T} = \mathbf{G} \mathbf{H} \mathbf{G}^T \rightarrow S \nabla_F \mathbf{T} S = S \mathbf{G} \mathbf{H} \mathbf{G}^T S = \mathbf{G} \mathbf{H} \mathbf{G}^T. \]

3.4 Triangular decomposition of symmetric Cauchy–like matrices
For the discussion of this section we will start with the displacement representation of a symmetric Cauchy–like matrix $\mathbf{C} \in \mathbb{R}^{n \times n}$,
\[ \Lambda \mathbf{C} - \mathbf{C} \Lambda = \mathbf{G} \mathbf{H} \mathbf{G}^T, \tag{13} \]
where $\Lambda$ is diagonal, $\mathbf{G} \in \mathbb{R}^{n \times n}$ is the generator, and $\mathbf{H} \in \mathbb{R}^{n \times r}$ is the signature matrix. In this representation of a general Cauchy–like matrix we have used the same letters for convenience.
Generally, the displacement representation (13) arises from other displacement representations, like i.e. the displacement representation of a symmetric Toeplitz matrix or another symmetric structured matrix like it was shown in the previous subsections. Matrix $C$ is not formed explicitly in order to avoid computational cost. Matrix $C$ is implicitly known by means of the generator pair. A triangular decomposition algorithm of $C$ works on the generator pair and it can be derived in an easy way as follows.

From equation (13), it is clear that any column $C_{\cdot,j}$, $j = 0, \ldots, n - 1$, of $C$ can be obtained by solving the Sylvester equation

$$AC_{\cdot,j} - C_{\cdot,j}\lambda_{j,j} = GHG_j^T,$$

and the $(i,j)$th element of $C$ can be computed as

$$C_{i,j} = \frac{G_iHG_j^T}{\lambda_{i,i} - \lambda_{j,j}}, \quad \text{iff } \lambda_{i,i} \neq \lambda_{j,j}$$

that is, for all elements off the main diagonal and maybe other elements. If $C$ is the Cauchy–like matrix $C = STS$ for $T$ a symmetric Toeplitz matrix, only the elements off the main diagonal can be computed by means of (14). For the case of Block-Toeplitz matrices $T$ (5) elements $t_{i,j}$, for $|i - j| \mod nb = 0$, that is elements of the diagonal of each block, can not be computed by means of (14). These last elements of $C$ must to be computed prior to the start of the factorization in another way that can be found in [21].

We assume in the following that all elements of $C$ are known or can be computed with (14). Let be the following partition of $C$ and $\Lambda$,

$$C = \begin{pmatrix} d & c^T \\ c & \hat{C} \end{pmatrix}, \quad \Lambda = \begin{pmatrix} \lambda & 0 \\ 0 & \hat{\Lambda} \end{pmatrix},$$

then, we define the following matrix $X$,

$$X = \begin{pmatrix} 1 & 0 \\ l & I \end{pmatrix}, \quad X^{-1} = \begin{pmatrix} 1 & 0 \\ -l & I \end{pmatrix},$$

where $l = c/d$. Let us pre-multiply by $X^{-1}$ and post-multiply by $X^{-T}$ the displacement equation (13),

$$X^{-1}(AC - CA)X^{-T} =$$

$$(X^{-1}AX)(X^{-1}CX^{-T}) - (X^{-1}CX^{-T})(X^TAX^{-T}) =$$

$$\begin{pmatrix} \lambda & 0 \\ \hat{\Lambda} - \lambda & \hat{\Lambda} \end{pmatrix} \begin{pmatrix} d & 0 \\ 0 & C_{sc} \end{pmatrix} - \begin{pmatrix} d & 0 \\ 0 & C_{sc} \end{pmatrix} \begin{pmatrix} \lambda & \hat{\Lambda} \\ 0 & \hat{\Lambda} \end{pmatrix} =$$

$$(X^{-1}G)H(X^{-1}G)^T,$$

where $C_{sc} = \hat{C} - (c^T)/d$ is the Schur complement of $C$ with respect to $d$. At this step, we have the first column of $L$, $(1 \quad l)^T$, and the first diagonal entry
of \( D, d \), of the \( LDL^T \) decomposition of \( C = LDL^T \) for a lower unit triangular factor \( L \) and a diagonal factor \( D \).

Equating the \((2, 2)\) position in the above equation, we have the displacement representation of the Schur complement of \( C \) with respect to its first element,

\[
\hat{A}C_{sc} - C_{sc}\hat{A} = G_1HG_1^T,
\]

where \( G_1 \) is the portion of \( X^{-1}G \) from the second row down. The process can now be repeated on the displacement equation of the Schur complement \( C_{sc} \) to get the second column of \( L \) and second element of \( D \) of the \( LDL^T \) factorization of \( C \). Repeating this process it can be obtained after \( n \) steps the \( LDL^T \) factorization of \( C \).

The steps described above are summarized in the following algorithm.

**Algorithm 1** \((LDL^T \) decomposition of symmetric Cauchy–like matrices.) \( \) Let \( G \in \mathbb{R}^{n \times r} \) be the generator, \( H \in \mathbb{R}^{r \times r} \) be the signature matrix, \( \lambda \) be an array with the diagonal entries of \( \Lambda \in \mathbb{R}^{n \times n} \) of the displacement of a symmetric Cauchy–like matrix \( C \) of the form \((13)\) and the entries of \( C \) that can not be computed by means of \((14)\), this algorithm returns a unit lower triangular matrix \( L \) and the diagonal entries of a diagonal factor \( D \), stored in the diagonal of \( L \), of the \( LDL^T \) decomposition of \( C \).

for \( k = 0, \ldots, n-1 \)
1. for \( i = k, \ldots, n-1 \)
   if \( \lambda_i \neq \lambda_k \)
   \( c_{i,k} = (G_i,HG_k^T)/(\lambda_i - \lambda_k) \).
   else
   (Value \( c_{i,k} \) is an entry to this algorithm).
   end if.
end for.
2. \( d = c_{k,k} \).
3. for \( i = k+1, \ldots, n-1 \)
   \( l_{i,k} = c_{i,k}/d \).
end for.
4. \( l_{k,k} = d \).
5. for \( j = 0, \ldots, r-1 \)
   for \( i = k+1, \ldots, n-1 \)
   \( g_{i,j} = g_{i,j} - g_{k,j}l_{i,k} \).
end for.
end for.
6. for \( i = k+1, \ldots, n-1 \)
   for \( j = k+1, \ldots, n-1 \)
   if \( \lambda_i = \lambda_j \)
   \( c_{i,j} = c_{i,j} - d l_{\max(i,j),k}^2 \).
   end if.
end for.
end for.
The $r$ columns of the generator are updated from the $(i+1)$th row down in step 5 of Algorithm 1. The $i$th row of the generator is not needed to update. The updating of such row gives a zero row and will not be reference in the next iterations.

Entries of $C$ that can not be computed by means of (14) must be updated before the next iteration as the generator. This is performed in step 6 of Algorithm 1. For example, if $C = STS$ for a symmetric Toeplitz matrix $T$, only the entries $c_{j,k}$, for $j,k > i$ and $j = k$, are needed to be updated, that is, step 6 involves only $(n-i)$ floating point operations in this case.

Algorithm 1 also returns the $i$th row of the generator in the $i$ iteration. Thus, on the return of the algorithm we will have a $n \times r$ matrix in which the $i$th row is the first row of the generator of the displacement representation of the $i$th Schur complement of $C$ with respect to its principal leading submatrix of order $(i-1)$. This will be useful for the blocking algorithm used in the sequential as in the parallel algorithm described in Section 4.2.

### 3.5 Implementation of the sequential algorithm

In this section we focus in a more computational point of view of Algorithm 1 and its blocking version. The implementation of the sequential algorithm is composed by several basic routines, some of them will be used in the parallel algorithm as well.

The first basic routine, called $dcol$, performs the computation of one column of the triangular factor $L$. The routine receives a piece of size $(n-k+1) \times r$ of the generator, being $r$ the number of columns of the generator, and one row (the $k$th row) of it, then, it computes the $k$th column of $n-k+1$ entries of the triangular factor. Fig. 2 shows the normal use of this routine to compute the $k$th column of $L$. Dashed lines represent not referenced matrix pieces.

However, this routine is prepared to be used in a more general way that lets to use it in a block–fashion as it will be further explained. Fig. 3 shows that it is possible to compute another set of entries, i.e. $m$ entries, of the $k$th column. These entries are marked with a bold–face line like the $k$th row of the generator. This last row will be known as the pivot row.

The following routine uses $dcol$ to compute a triangular factor. The triangular factor computed by $dtrf$ can only be one triangular piece of the total triangular factor $L$ as it can be seen in Fig 4.

Subroutine $dupd$ is very similar to $dtrf$ due to it uses the basic routine $dcol$. $dupd$ allows to compute a rectangular block of $L$ with the desired number of rows and which number of columns are the same as the order of the previously computed triangular block situated over. Fig 5 shows an example of the rectangular block (with bold–face lines) computed by this routine. In order to compute this rectangular block ([$B$]) it is needed to use the rectangular block of the generator, denoted as $V$, that gives rise to the triangular block denoted by...
3 THE SEQUENTIAL ALGORITHM

$\begin{bmatrix} r \\ \end{bmatrix}$

$n-k+1$

Figure 2: Computation of the $k$th column of the triangular factor $L$ by means of the $dcol$ subroutine

$L$ in the figure, in addition to the piece $G$ of the generator. It is assumed that the triangular block $L$ is computed previously to the computation of $B$.

The following two subroutines use the above explained ones in order to build blocking algorithms for the computation of all the triangular factor $L$. The first one, $dupdx$, makes exactly the same function as $dupd$ but partitioning the computed rectangular block $B$ (Fig. 5) in smaller pieces. Each one of these pieces are computed by means of $dupd$. The size of $nb$ is machine dependent and must be tuned before running the overall algorithm though it will be a fixed size for each processor so it is only needed to tune once.

$\begin{bmatrix} r \\ \end{bmatrix}$

$\begin{bmatrix} m \\ \end{bmatrix}$

Figure 3: Computation of a piece of $m$ entries of the $k$th column of the triangular factor $L$ by means of the $dcol$ subroutine
The other blocking routine is dtrfx. This is the last routine needed to obtain the triangular factor $L$. dtrfx is the driver routine called to obtain $L$. It uses dtrf and dupdx alternatively as it is shown in Fig. 6, where the numbers represent the order in the computations. Odd numbers represent triangular factors computed by dtrf whereas even numbers represent rectangular factors computed by dupdx. The size of the triangular blocks is also machine dependent and it is tuned as in the dupdx routine. With this last routine we exploit as much as possible the lower level of memory, that is the cache memory, in order to
have the most competitive sequential algorithm to work the parallel algorithm with.

This blocking partition is also important because a similar way of partition and distribution of the workload will be used in the parallel algorithm proposed in the following sections of this work. Finally, it is important to note that the asymptotical cost of `dupdx` is of $O(n^2)$ floating point operations and $O(n^2/2)$ operations for the `dtrfx` routine, that is, the computation of a triangular block is half the time used for the computation of a square block. This costs will be used in the construction of the performance model, described in Section 4.3, of the heterogeneous algorithm.

4 The Parallel Algorithm, the HeHo strategy

The HeHo strategy for the design of parallel algorithms on Heterogeneous NoC’s uses a Heterogeneous distribution of processes over processors and Homogeneous block distribution of data over the processes. This is carried out by mapping different number of processes over the physical processors according to their performance. The tool that allows to do that is HeteroMPI. Specifically, HeteroMPI is a programming environment that, working together with the mpC programming environment, allows to translate a homogeneous parallel algorithm based on calls to BLACS/ScaLAPACK routines to a heterogeneous environment of computation without changing anything in the original homogeneous algorithm. However, this step is not easy due to a very accurate computational model of the algorithm is needed to be written in order the mapping runtime system to map the processes in the best way to achieve the maximum performance in the
heterogeneous NoC’s.

The following two subsections are devoted to explain the homogeneous parallel algorithm. Section 4.3 represents the central part of this section in which the performance model of the parallel algorithm is deeply described. More practical details are explained in the following section and experimental results are also shown at the end of this section.

4.1 The Parallel Algorithm

The parallel algorithm for the solution of the least squares problem consists of three main steps:

1. “Previous computations”.
2. Triangular decomposition.
3. Triangular system solution.

Very briefly, “Previous computations” refers to the construction of the generator matrix on which the triangularization algorithm (second step) works plus some auxiliary operations. The third step refers to the solution of two triangular systems that leads to the final solution of the normal equations.

The main part of the algorithm on which the most computational work falls is the triangularization decomposition (second step) since it represents between 80% and 90% of the overall cost. Next, we will focus our attention only in this second step.

4.2 Parallel triangularization of symmetric Cauchy–like matrices

In this section it is described the parallel implementation of the sequential algorithm (Algorithm 1) described in sections 3.4 and 3.5. With the description used in such sections the parallel algorithm can be naturally described and easily understood.

Cauchy–like matrices, according we are using them, are implicitly known by means of the generator and the displacement matrices \((G, H)\) as it can be seen in its displacement representation

\[
C = GHG^T.
\]

Due to this fact we can derive the parallel algorithm for a general symmetric Cauchy–like matrix \(C\) independently of how it has been formed. For the parallel triangularization of \(C\),

\[
C = LDL^T,
\]

where \(L\) is unit lower triangular and \(D\) is diagonal, there are two main factors in the data distribution that we can use,

- by definition of structured matrix it is expected that \(n \gg r\), and
the operations performed in the triangularization process can be carried out independently on each row of the generator.

Really, there exists a particularity in the symmetric case concerning the restriction that supposes not to be able to compute directly certain entries of the triangular factor \( L \) by means of the main operation, as it was mentioned in Section 3.4, which implies that these entries must be computed by a different operation. In order to simplify the exposition we will obey this detail since it does not affect the main issue of the exposition.

First of all, the generator of size \( n \times r \) is divided into blocks of size \( nb \times r \), that is, a block size of \( nb \) rows is chosen in order to make the following partition of the generator,

\[
G = \begin{pmatrix}
G_0 & \quad & \\
G_1 & \quad & \\
\vdots & \quad & \\
G_{n/nb-1}
\end{pmatrix},
\]  

(17)

These blocks are cyclically distributed among processes so \( G_i, i = 0, \ldots, n/nb - 1 \), belongs to process \( P_{i \mod p} \). For simplicity in the exposition we will assume in the next \( (n \mod nb) = 0 \) although this condition have not to be accomplished in the implementation.

The unit lower triangular factor \( L \) (16) obtained is partitioned in a two dimensional array of \((n/nb) \times (n/nb)\) square blocks of order \( nb \),

\[
L = \begin{pmatrix}
L_{0,0} & L_{1,1} & \cdots \\
L_{1,0} & L_{2,2} & \cdots \\
\vdots & \vdots & \ddots \\
L_{n/nb-1,0} & L_{n/nb-1,1} & \cdots & L_{n/nb-1,n/nb-1}
\end{pmatrix},
\]  

(18)

where each square block \( L_{i,j} \), for \( j = 0, \ldots, (n/nb - 1) \), belongs to processor \( P_{i \mod p} \) as the generator blocks. In Fig. 7 it can be seen an example of distribution of both \( G \) and \( L \) in a “logical column” of three processors.

The BLACS distribution model used to manage logically distributed arrays highly helps to build a distribution like the one described for our parallel algorithm. A simply “logical column” of processes will be sufficient.

The diagonal matrix \( D \) (16) is stored in the diagonal entries of \( L \) since all the diagonal entries of \( L \) are implicitly one.

The block triangular factorization is a finite iterative process that can be described as follows (Fig. 8). Let \( k = 0, \ldots, (n/nb - 1) \) denote the iteration number then, before the execution of the \( k \) iteration, the \( G_i \) blocks, \( i < k \), have been zeroed and the \( L_{i,j} \) blocks, such that \( j < k \) for all \( i \), have been computed.

At iteration \( k \), processor containing the \( G_k \) block computes \( L_{k,k} \) and zeroes \( G_k \) using the algorithm described in Section 3.4. At once, the processor broadcasts the suitable information to the rest of processors. The rest of the processors receive the said information. Processors containing the \( G_i \) blocks, \( i > k \), compute
the $L_{i,k}$ blocks, $i > k$, and update the $G_i$ blocks. Blocks $L_{i,j}$, $j > k$ for all $i$, have not been referenced yet at this time. Then the algorithm goes on the next iteration.

The value $nb$ chosen has a great impact in the parallel algorithm. High values of $nb$ produce low number of messages of big size but the load is unbalanced in this case. Low values of $nb$ produces higher number of messages with lower size with better load balancing. Furthermore, this parameter highly depends on the hardware platform so it must to be chosen by experimental tuning.

The operation of the parallel algorithm can be easily seen by following the ordering numbers in Fig. 9. The order is exactly the same as the one shown in Fig. 6. The main different now is that the rectangular blocks are computed concurrently due to the existing intrinsic parallelism at row operation level.

The following is the main piece of code of subroutine pdtrf, the analog parallel subroutine to the sequential one dtrfx described in Section 3.5.

```
1 *
2  call blacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )
3 *
4  if( nprow.eq.1 ) then
```
Figure 9: Order of the computation of the blocks that form the triangular factor \( L \) by means of the \texttt{pdtrf} subroutine. Example with 3 processors.

```c
5 * call dtrfx( n, r, G, lld, L, lld )
7 return
8 *
9 end if
10 *
11 do k = 1, n-nb, nb
12 *
13 call infog1l( k, nb, nprow, myrow, rsrc, lk, pkrow )
14 *
15 if( myrow.eq.pkrow ) then
16 *
17 call dtrfx( nb, r, G( lk ), lld, L( lk+k*lld ), lld )
18 call dlapay( 'A', nb, r, G( lk ), lld, V, nb )
19 *
20 call dgebs2d( ictxt, 'C', ', ', nb, r, V, nb )
21 *
22 else
23 *
24 call dgebr2d( ictxt, 'C', ', ', nb, r, V, nb, pkrow, 0 )
25 *
26 end if
```
First of all, the subroutine calls to `blacs_gridinfo` (line 2) in order to obtain the coordinates of the calling process (`myrow` and `mycol`) together with the shape of the logical grid (`nprow` and `npcol`) on which the parallel algorithm runs, for a given context manager number `ictxt`. Obviously, `mycol=npcol=1` in our case.

Next, if there is only one process in the network (line 4) this process calls subroutine `dtrfx` to solve the problem sequentially (line 6). `dtrfx` receives a generator `G` of size `n×r` and returns the triangular factor `L` of order `n`.

The general case when the number of processes is greater than only one is solved by means of the loop between lines 11 and 35. This loop basically computes a triangular block and the rectangular block down the computed triangular block in each iteration, that is, iteration `k` computes blocks marked as `k` and `k+1` in Fig. 9. The size of the block `nb` is the block size `nb` used in the theoretical exposition.

The implementation of the algorithm uses the style of BLACS and ScaLAPACK routines as it can be seen by the use of the `infog1l` routine (line 13). Given a global index `k` of the distributed matrix `G`, the block size `nb`, the number of processes in one dimension `nprow`, the coordinate of the calling process in this dimension `myrow` and the coordinate of the sources process in the same dimension that has the first element of the distributed array `rsrc`, this routine returns the index of the global index `k` in the local memory `lk` and the coordinate of the owner processor (`pkrow`) of the global `k`th element. A more detailed description can be obtained in the code of the routine itself. `infog1l` returns all needed information to proceed with the rest of the computations.

The process owner of element `k` (lines 15–22) is in charge to compute the triangular block whereas the others only have to wait for data from process...
pkrow. Process pkrow computes the triangular factor calling dtrfx, stores the nb x r rectangular piece of the generator G used to compute de triangular factor in a different place of memory (V) (line 18) and broadcasts V to the rest of processes (line 20). The rest of processes receive V (line 24). BLACS routines dgebs2d and dgebr2d are enough to perform the intercommunication operation and offers an easy interface over MPI to manage matrix communication operations.

The rest of the code until the end of the loop is devoted to the computation of the squares blocks down the triangular block that has just been computed. The auxiliary routine numroc (line 31) lets to obtain the total amount of rows of the rectangular block that must been computed by the calling process. The computation is carried out by means of the routine dupdx (lines 32 and 33) described in Section 3.5.

Lines 37–44 are for the computation of the non-square blocks if they exist, that is, if (n mod nb) ≠ 0. As it has been mentioned above, details about the computation of certain entries of the triangular factor L are omitted here for clarity though the final algorithm obviously takes into account these computations.

Finally, it can be easily shown that the communication cost of the parallel algorithm is of nb x r floating point scalars per iteration. A useful data that will be used in the design of the performance model.

4.3 The Performance Model

The cornerstone of the connection between a SPMD program consisting of several MPI processes and the program itself running on a Heterogeneous NoC’s is the Performance Model. The Performance Model is based on the notion of network as it is used in the context of the mpC language introduced by A. Lastovetsky [1]. A network corresponds to a group of processes jointly performing some parallel computations. A mpC network is an abstraction facilitating the work with actual processes of the parallel program. Firstly, the programmer must define a network consisting of a given number of abstract processors, and then describe the parallel computations on this network. Abstract processors representing the network will be mapped in real processors one to one of the physical NoC’s according to the performance description of the behaviour of the parallel algorithm. Therefore, the Performance Model must define the mpC network with sufficient detail so the mapping algorithm can correctly map the program processes (represented by the abstract processors of the network) in the suitable real processor to achieve the maximum performance.

The mpC language is one of the best known tools to implement parallel applications for Heterogeneous NoC’s. However, mpC is based on the used of processes that work with different amount of data, that is, unbalanced processes so the mapping algorithm fits only one process per real processor. Working with mpC does not allow to reuse existing parallel code for homogeneous NoC’s. In this scenario appears a very useful tool also introduced in [1] and widely developed by Ravi Reddy in [22]: the Heterogeneous Message Passing Interface (HeteroMPI).
HeteroMPI is an adaptation of the mpC language to the MPI programming level. HeteroMPI automates the selection of a group of processes that executes the heterogeneous algorithm faster than any other group. The algorithms used to solve the problem of process selection are essentially the same as those used in the mpC compiler. HeteroMPI has an analog routine to the MPI one for creating groups of processes. During the creation of the processes, the HeteroMPI runtime system solves the problem of selecting the optimal set of processes running on different computers of the heterogeneous networks. Summarizing, HeteroMPI lets to run MPI applications on a heterogeneous network balancing the load by choosing the suitable number of processes that will run on each real processor according to the physical features of it and the performance model defined for the parallel program.

The following code corresponds to the performance model of our parallel algorithm.

```c
1 double cost( int n, int nb, int p, int I );
2
3 nettype NetType(int n, int r, int nb, int nbb, int p)
4 {
5     coord I=p;
6     node {
7         I>=0:
8         bench * cost( n, nb, p, I ) * ( nb*nb / (double) (nbb*nbb) );
9     }
10     link ( J=p )
11     {
12         I>=0 && J!=I:
13             length * ( (nb*r)*(n/(nb*p)+(n%nb?1:0))*sizeof(double) )
14             [I] -> [J];
15     }
16     parent [0];
17     scheme
18     {
19         int P, k, i, j;
20         double cc, cl;
21         int nbkks = n/nb;
22         double propor;
23         cl = (n/(nb*p)+(n%nb?1:0));
24         nbkks = nbkks + ((n%nb)!=0);
25         for( k = 0; k < nbkks; k++) {
26             P = k%p;
27             cc = cost( P, p, nb, n );
28             propor = 1.0;
29             if( k==nbkks-1 ) propor = n%nb / (double) nb;
```
4 THE PARALLEL ALGORITHM, THE HEHO STRATEGY

\[(100.00 \cdot \text{propor} \cdot \text{propor} / \text{cc}) \% \[ P \];
\]

\[
\text{par}(i = 0; i < p; i++)
\]

\[
\text{if}(i!=P)
\]

\[
(100.00/c1) \% \[ P \] -> [ i ]; \quad \text{// Broadcast}
\]

\[
\text{par}(i = 0; i < p; i++)
\]

\[
\text{cc} = \text{cost}(i, p, \text{nb}, n);
\]

\[
\text{for}(j = k+1; j < \text{nblks}; j++)
\]

\[
\text{if}(j\%p == i)
\]

\[
\text{propor} = 1.0;
\]

\[
\text{if}(j==\text{nblks}-1) \text{propor} = n\%\text{nb} / (\text{double}) \text{nb};
\]

\[
(2.0 \cdot 100.00 \cdot \text{propor} / \text{cc}) \% \[ i \]; \quad \text{// Computing square blks}
\]

\[
\text{for}(i = 0; i < \text{nblocks}; i++)
\]

\[
\text{if}(i\%p == I)
\]

\[
\text{c} = \text{c} + 2.0 \cdot i + 1.0;
\]

\[
\text{propor} = n\%\text{nb} / (\text{double}) \text{nb};
\]

\[
\text{if}(\text{propor}>0.0 \&\& (\text{nblocks}\%p)==I)
\]

\[
\text{c} = \text{c} + \text{propor} \cdot (2.0 \cdot i + \text{propor});
\]

\[
\text{return c};
\]

The first item in the network definition corresponds to the association of the abstract processors with a coordinate system (line 6). Each abstract processor in the network is identified in our performance model by an integer \( I \) representing its coordinate in a line of \( p \) processors ranging from 0 to \( p-1 \).

The next lines 7–10 describe the total amount of computation performed by all of each abstract processors \((I>0)\). The runtime mapping algorithm
included in the mpC environment performs a benchmark operation whose time is represented by variable $\text{bench}$. This benchmark corresponds to the computation of a triangular block of order $\text{nbb}$ by means of $\text{dtrfx}$. The value $\text{nbb}$ is fixed by the main program and passed to the description network. The same is used for all different problem sizes for which is used this program. The time returned in the $\text{bench}$ variable is used to estimated the real computational cost of the $I$th abstract processor. Due to the complexity of the analytical formulas that describe the total amount of computational cost performed by each process, we have used the function $\text{cost}$ to compute this cost (lines 52–70).

The first loop of $\text{cost}$ counts the number of square blocks multiplied by 2 plus the number of triangular blocks of the triangular factor belonging to each processor. Both type of blocks are of order $\text{nb}$, that is, the block size $\text{nb}$ used to make the partition of the generator (17) and the triangular (18) factors as it can be seen in the example shown in Fig. 7. Fig. 10 shows a distribution example for a problem of size $n = 27$ with three processors and a block size of 5. In this example, the first loop of function $\text{cost}$ returns 8 when calling by $P_0$, that is, 3 squares blocks $\times$ 2 plus 2 triangular blocks $\times$ 1. And also returns 12 and 5 for $P_1$ and $P_2$, respectively. The rest of $\text{cost}$ returns the proportional part of an incomplete row of blocks as it happens in the example. In the example, this last part returns a value of 4.16 when is called by $P_2$ and 0 otherwise. Really, function $\text{cost}$ returns 8, 12 and 9.16 when calling by $P_0$, $P_1$ and $P_2$, respectively.

Summarizing, the total amount of computation performed by an abstract processor and described in line 9 is the result of the product of the benchmark time for computing a triangular block of size $\text{nbb}$ ($\text{bench}$), the number of times that an abstract processor performs a computation equivalent to the computation of a triangular block of size $\text{nb}$ ($\text{cost}(\text{n}, \text{nb}, \text{p}, \text{I})$), and the proportion between asymptotical cost of a $\text{nb}$ block regarding a $\text{nbb}$ block.

The next section of the performance model describes the total communication cost (lines 11–16). This cost corresponds to a broadcast per iteration of blocks of $\text{nb} \times \text{r}$ double precision scalars. Line 14 shows the physical link performance returned by the runtime environment ($\text{length}$) multiplied by the total amount of broadcasts performed by processor $I$ to processor $J$.

Until line 17 all the basic performance model of the parallel algorithm is described through its total costs, computational ($\text{node}$) and communication ($\text{link}$). However, as our preliminary results shown, this description is not enough to the mapping algorithm to distribute optimally the workload, that is, to choose the suitable number of processes per processor. The next part of the performance model description shows the parallel algorithm behaviour under the point of view of the order and the cost of the different operations performed.

The scheme description for this case is made up for a main loop. Each iteration of this loop corresponds to the computation of a triangular block. Processor $P$ is the owner of the triangular block computed in iteration $k$. Line 32 is the percentage of the total computational cost computed by processor $P$ when computes only this triangular block. Next lines 33–37 describe the communication percentage corresponding to the broadcast of a rectangular block of size $\text{nb} \times \text{r}$. It is described by means of a parallel loop due to the communication from one
to all processors is concurrently. Lines 38–47 describe the computation of the squares blocks down the computed triangular one just computed in this $k$th iteration. These blocks are computed concurrently by each processor so the loop indexed by variable $i$ is a parallel structure. Inner loop ($j$) goes over each of these squares and, only for these ones owned by the calling processor are computed. Line 44 shows the percentage of the total amount of computation performed by processor $i$ devoted to the computation of the square block $j$.

### 4.4 Implementation of the Parallel Algorithm

In this section is described the main routine of the parallel algorithm. This routine is written in C. The main feature of the programming model followed is shown in Fig. 11. The Hetero MPI provides with the facility of using the homogeneous model of computation in the real running environment of a heterogeneous NoC. This is done by means of performing certain operations before calling the main driver routine that solves the problem, that is, the triangularization of a Cauchy–like matrix in our case. We will show the main operations through the description of the main program.

First of all, the main routine works in the HeteroMPI environment as it ...
can be seen in Fig. 11. The first operation deals with the performance of the underlying physical processors over the parallel algorithm is going to work. All starting processes, belonging to the predefined communication universe, known as \texttt{HMPI\_COMM\_WORLD\_GROUP}, call to the \texttt{HMPI\_Recon} routine. This routine performs a call to a benchmark routine chosen by the programmer. In our case, the benchmark routine is \texttt{dtrfx}, that is, the routine that allows to obtain a triangular block as described in Section 3.5. The size of the block of the generator on which the benchmark routine works is $nbb \times r$, where $nbb$ is a parameter that can be passed to the main program or also can be a fixed size, and $r$ is the number of columns of the generator. The number $nbb$ needs to be chosen as large as the time spent by the benchmark function could be used to accurately estimate the relative performance of the different physical processors and, at the same time, as little as possible to avoid interferences in the main result. This choice is very important because more or less fine mapping depends on it.

The main driver routine, \texttt{pdtrfx}, depends on several parameters. These parameters are not only parameters defined by the problem itself like the generator size ($n \times r$), but other additional parameters like the block size ($nb$) and the number of processes. This fact means that these last parameters would be tuned before calling the parallel routine in order to both execute it with the best values of them and to release the user for manually choose them, whom would force the user to have a very knowledge of the parallel algorithm and
The next step in the parallel algorithm deals with the tuning of the number of processes and the block size $nb$. Two nested loops indexed by attempted values perform this computation by calling to the `HMPI_Timeof` routine. This routine makes an estimation of the total time of the parallel algorithm without real computation. Execution time for `HMPI_Timeof` is negligible and uses the information provided with the performance model. After the execution of this algorithm, the best values are obtained and the parallel algorithm will work with this choice. In the next section it is shown an example of the behaviour of this tuning algorithm.

Really, the HeteroMPI environment setup starts next. This setup consists of similar steps to the MPI environment setup. The host processor, that is, the parent processor under the mpC terminology, calls routine `HMPI_Group_create` with the suitable arguments to create a HeteroMPI work group of abstract processes. The number of processes are the ones chosen by the previous tuning algorithm ($p$). The rest of processes calls the same routine as well. This must be done in this way since `HMPI_Group_create` is a collective operation. The HeteroMPI group will be composed by only this $p$ processes so the rest of processes that were started when the parallel program were called must exit the application calling `HMPI_Finalize`. Thus, only the chosen $p$ processes will continue to the end of the execution.

Before these HeteroMPI operations the MPI environment takes place in order to have the possibility of running the homogeneous routine as it is shown in Fig. 11. The link between HeteroMPI and MPI environments is made up by a call to the HeteroMPI routine

$$\text{mpicomm} = *\text{(MPI_Comm*)HMPI_Get_comm(&gid)};$$

This routine allows to obtain a MPI communicator (`mpicomm`) from a HeteroMPI group of processes identified in the example as variable `gid`. This is the only step representing the MPI environment due to the rest of the parallel algorithm uses the BLACS model.

In the next step the BLACS environment is setup as usual in the programming with homogeneous NoC's. The connection between the MPI and BLACS code is performed by

$$\text{ictxt} = \text{Csys2blacs_handle(mpicomm)};$$

where `ictxt` represents the BLACS context. Having this context the following code corresponds to a typical BLACS/SeaLAPACK code by initializing the logical BLACS grid (`gridinit`) and so on.

Each of the nested environments finish by calling the appropriate closing routines like `gridexit` for the BLACS context. The HeteroMPI environment finishes the execution by calling `HMPI_Group_free` to destroy the HeteroMPI group and `HMPI_Finalize` at the end of the program.

The above description shows how is possible to use a homogeneous parallel algorithm as the one described in sections 4.1 and 4.2. At the same time,
Table 1: Main characteristics of the processors of the Heterogeneous cluster

<table>
<thead>
<tr>
<th>Name (number of processors)</th>
<th>Architecture</th>
<th>cpu MHz</th>
<th>Total Main Memory (mBytes)</th>
<th>Cache (kBytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg1clusterXX (2)</td>
<td>Linux 2.6.8-1.521smp Intel(R) XEON(TM)</td>
<td>2048</td>
<td>1024</td>
<td>512</td>
</tr>
<tr>
<td>csultraXX (1)</td>
<td>SunOS 5.8 sun4 sparc SUNW,Ultra-5_10</td>
<td>440</td>
<td>512</td>
<td>2048</td>
</tr>
</tbody>
</table>

we have shown through the steps followed to move from a Homogeneous to a Heterogeneous environment need a detailed and non-trivial study of both the parallel program and the tools provided by the mpC/HeteroMPI programming and runtime environments.

4.5 Experimental Results

The experimental results have been taken in a Heterogeneous NoC’s composed by two homogeneous clusters, each one composed by a different processor. Table 1 shows the individual main characteristics of processors of each cluster. The pg1cluster is composed by 4 processors of the first type whereas csultra cluster is composed by 7 processors, each processor has a different number within the cluster at the end of its name (representing by XX).

The first result concerns the tuning algorithm to test the optimum values with the parallel algorithm must run. As it is shown in the previous section, this algorithm runs previously to call the main routine. For this test we have chosen a generator matrix of 7920 \times 100. The sequential time needed to solve the problem with this size in the fastest processor, that is, in an Intel processor is 27 seconds.

The Virtual Parallel Machine (VPM) used for this test is composed by one two-processor Intel’s board and 7 Ultra Spark processors (Fig. 12). Each node has a total amount of concurrent processes that can host, represented by the first number. The second number between square brackets represent the number of physical processors per machine. We have chosen these numbers for the maximum number of processes due to Intel processors is theoretically \approx 4 times faster.

Running the tuning algorithm based on the execution of the \texttt{MPI\_Timeof} routine, we have the time results shown in Fig. 13. Each graphic represents time regarding different block sizes for different number of processes. Based on these results, the tuning algorithm chooses 11 processes and a block size of 22. With this choice, the parallel algorithm returns its result in 11.419 seconds. We have carried out a large amount of tests manually varying these both parameters, the number of processes and the block size. Our results confirm that
the tuning algorithm chooses quite accurately both parameters estimating the execution time. More processes than 11 have offered poorer results because of the overhead involved of more processes inside a two-processor board or/and no more Ultra Sparc machines have been selected due to the overcost induced by the communication cost. The runtime mpC environments maps 8 processes on the two Intel processors and three processes on each of three Ultra Sparc's.

5 The Parallel Algorithm, the HoHe strategy over HeteroMPI

The HeHo strategy followed in Section 4 uses a Heterogeneous distribution of processes over processors and Homogeneous block distribution of data over the processes providing HeteroMPI the necessary tools to work under this strategy. On the contrary, the HoHe strategy uses a Homogeneous distribution of processes over processors with each process running on a separate processor and Heterogeneous block cyclic distribution of data over the processes. mpC is the tool to work under this second strategy. Basically, the HoHe strategy is based on a program that manage data distribution by means of mpC making calls to LAPACK. The HeHo strategy, as we have seen before, is mainly a heterogeneous program using calls to ScaLAPACK where the processes can be unevenly distributed among the physical processors. In the HoHe strategy, only one process is mapped onto one processor so there exists different amount of work for each process. The different amount of computation comes from the different amount of data each process works with.

In this section we propose a new approach to the HoHe strategy different from the one followed by mpC. Our purpose is to establish a model for a likely future HeteroScaLAPACK. Our intention is to implement HoHe parallel programs, that is, one process per processor an different amount of computation for the different processes. Really, all processes are essentially the same since we keep using the SPMD programming model but processes perform the same computations with different amount of data.
Our model is based on the modification of PBLAS and ScaLAPACK. However, a small number of modifications introduced by the model make this translation of homogeneous to heterogeneous PBLAS/ScaLAPACK routines easy to carry out. No modifications are needed in the underlying BLACS and MPI environments over PBLAS/ScaLAPACK routines work. A heterogeneous programming and runtime environments are still needed, but the available mpC/HeteroMPI provide the programmer with enough tools to perform this task so no further tools or tedious learning will be required from those programmers that usually work with PBLAS/ScaLAPACK.

In the next sections we will introduce this new model through the example program developed in the previous ones. Although there exist some restrictions in the example we will show our main aim of our idea: easiness and efficiency.

### 5.1 Data distribution

The first step to introduce our HoHe model through the example consists of the description of the heterogeneous data distribution of our example. Data will be unevenly distributed over the processes as it can be seen in Fig. 14.

Our model consists of the following ideas:

- The processes are arranged in a bidimensional logical grid as the BLACS computational model sets. Each process is identified by the column and row coordinates.
- There exist a different block size per coordinate and process.
Blocks are distributed cyclically among the rows and columns of processes.

As it can be seen, the above items basically describe the BLACS model except for the different block sizes. Fig. 14 shows an example of data partition following this heterogeneous model in the particular case of a one-dimensional grid. Our problem force us to distribute data among one dimensional array of processes but it easy to see how it would be a distribution over a two-dimensional logical grid of processes if it is required.

Although there is not necessary in this example, is absolutely possible to use different block sizes for the rows and the columns. Fig. 15 shows an example of a two-dimensional heterogeneous distribution of a bidimensional array with our model. This figure represents a distributed matrix of $16 \times 25$ onto a logical array of $3 \times 4$ processes $P_{i,j}$, $i = 0, \ldots, 2$ and $j = 0, \ldots, 3$. Each process holds a different block size, i.e. $P_{2,3}$ stores blocks of size $6 \times 2$ or $P_{1,0}$ stores blocks of size $2 \times 5$.

In our example shown in Fig. 14 each process is assigned with a different block size, that is, $P_0$ is assigned with $\nu = 4$, $P_1$ with $\nu = 6$ and $P_2$ with $\nu = 10$. The distribution of block is cyclical so each process has two row blocks with a
5. THE PARALLEL ALGORITHM, THE HOHE STRATEGY OVER HETEROOMP32

![Diagram of data distribution]

Figure 15: Two-dimensional data distribution in an heterogeneous logical grid.

In order to further clarify the explanation we have added one more figure. Fig. 16 shows the same as Fig. 14 from the point of view of each process.

5.2 Modified ScaLAPACK Tools

The BLACS/ScaLAPACK parallel programming model is based on the Array Descriptor of distributed arrays. This array stores the information required to establish the mapping between each global array entry and its corresponding process and memory location. The length of the array descriptor is specified by DLEN_ and varies according to the descriptor type DTYPE_. Table 2 shows the content of the array descriptor entries for in-core dense matrices as it is explained in the User’s Guide [23].

It is clear that all processes share the same blocking size for rows and columns store in entries 5 and 6, respectively. There is only one block of size DESCTBM A × DESCNTBM A. In our heterogeneous programming model we need to represent different block sizes with the same techniques based on a descriptor array.

There exists two main approximations to modified the homogeneous descriptor in order to build a heterogeneous one:

- to represent only the different block sizes, or
- to represent the block size of each processor.
<table>
<thead>
<tr>
<th>DESC_( )</th>
<th>Symbolic name</th>
<th>Scope</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DTYPE_A</td>
<td>(global)</td>
<td>Descriptor type DTYPE_A=1 for dense matrices.</td>
</tr>
<tr>
<td>2</td>
<td>CTXT_A</td>
<td>(global)</td>
<td>BLACS context handle, indicating the BLACS process grid over which the global matrix A is distributed. The context itself is global, but the handle (the integer value) may vary.</td>
</tr>
<tr>
<td>3</td>
<td>M_A</td>
<td>(global)</td>
<td>Number of rows in the global array A.</td>
</tr>
<tr>
<td>4</td>
<td>N_A</td>
<td>(global)</td>
<td>Number of columns in the global array A.</td>
</tr>
<tr>
<td>5</td>
<td>MB_A</td>
<td>(global)</td>
<td>Blocking factor used to distribute the rows of the array.</td>
</tr>
<tr>
<td>6</td>
<td>NB_A</td>
<td>(global)</td>
<td>Blocking factor used to distribute the columns of the array.</td>
</tr>
<tr>
<td>7</td>
<td>RSRC_A</td>
<td>(global)</td>
<td>Process row over which the first row of the array A is distributed.</td>
</tr>
<tr>
<td>8</td>
<td>CSRC_A</td>
<td>(global)</td>
<td>Process column over which the first column of the array A is distributed.</td>
</tr>
<tr>
<td>9</td>
<td>LLD_A</td>
<td>(local)</td>
<td>Leading dimension of the local array. LLD_A&gt;MAX(1,LOC_r(M_A)).</td>
</tr>
</tbody>
</table>

Table 2: Content of the array descriptor for in-core dense matrices
The first choice only means that the descriptor will store the different block sizes, one per type, whereas the second choice involves to have a different entry per processor and dimension. We chosen the second option because, though it is necessary to have one entry per processor and dimension, makes easier the implementation of the parallel routines and requires fewer communications between processors than the other option due to all processes have the information contained in the descriptor array and, thus, know the block size of each other process.

The following is an example that would help to understand. The descriptor array for certain entries of distributed matrix shown in Fig. 15 has the entries shown in Table 3.

Now, the descriptor size is not a fix number, not depends on the different sizes of blocks but depends on the number of processes working in the NoC’s. For example, if there are only two types of processor but 2 and 6 instances of each one, it can be used the descriptors shown in Table 4. We assume that processors of the 6-set are two times faster than the two processors of the other set. The descriptors shown in the table corresponds to the arrays of Fig. 17.

The array descriptor designed in this way are not of a fixed size. The size

Figure 16: Distribution of Fig. 14 under the process point of view. $b_i$, $i = 0, \ldots, 5$ is the block number
Table 3: Content of the array descriptor of matrix in Fig. 15

<table>
<thead>
<tr>
<th>DESC_(.)</th>
<th>Symbolic name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DTYPE_A</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CTXT_A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>M_A</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>N_A</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>MB_A0</td>
<td>3</td>
<td>Rows of the distributed matrix.</td>
</tr>
<tr>
<td>6</td>
<td>MB_A1</td>
<td>2</td>
<td>Rows of block of ( P_{0,0 \leq j \leq 3} ).</td>
</tr>
<tr>
<td>7</td>
<td>MB_A2</td>
<td>6</td>
<td>Rows of block of ( P_{1,0 \leq j \leq 3} ).</td>
</tr>
<tr>
<td>8</td>
<td>NB_A0</td>
<td>5</td>
<td>Columns of block of ( P_{0 \leq j \leq 2,0} ).</td>
</tr>
<tr>
<td>9</td>
<td>NB_A1</td>
<td>3</td>
<td>Columns of block of ( P_{0 \leq j \leq 2,1} ).</td>
</tr>
<tr>
<td>10</td>
<td>NB_A2</td>
<td>7</td>
<td>Columns of block of ( P_{0 \leq j \leq 2,2} ).</td>
</tr>
<tr>
<td>11</td>
<td>NB_A3</td>
<td>2</td>
<td>Columns of block of ( P_{0 \leq j \leq 2,3} ).</td>
</tr>
<tr>
<td>12</td>
<td>RSRC_A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>CSRC_A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>LLD_A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Two possible descriptors for two different type sets of 2 and 6 processors, respectively

<table>
<thead>
<tr>
<th>DESC_(.)</th>
<th>Symbolic name</th>
<th>Value</th>
<th>Symbolic name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DTYPE_A</td>
<td>1</td>
<td>DTYPE_A</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>CTXT_A</td>
<td></td>
<td>CTXT_A</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>M_A</td>
<td></td>
<td>M_A</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>N_A</td>
<td></td>
<td>N_A</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>MB_A0</td>
<td>5</td>
<td>MB_A0</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>MB_A1</td>
<td>5</td>
<td>MB_A1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>NB_A0</td>
<td>2</td>
<td>MB_A2</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>NB_A1</td>
<td>4</td>
<td>MB_A3</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>NB_A2</td>
<td>4</td>
<td>MB_A4</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>NB_A3</td>
<td>4</td>
<td>MB_A5</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>RSRC_A</td>
<td></td>
<td>MB_A6</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>CSRC_A</td>
<td></td>
<td>MB_A7</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>LLD_A</td>
<td></td>
<td>NB_A0</td>
<td>5</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td>RSRC_A</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td>CSRC_A</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>LLD_A</td>
<td></td>
</tr>
</tbody>
</table>
of the descriptors depends on the number of processes, the topology of the grid and the different type of physical processors onto the distributed matrix is going to be mapped. The heterogeneous version of PBLAS and ScaLAPACK routines must take into account this fact. However, as it will be shown, this is more simple that it can be looks like.

5.3 Implementation of the Parallel Algorithm

Firstly, we are going to show the implementation of parallel triangularization. This algorithm was shown in Section 4.2. The code analog to the one shown in page 18 has the following form.

```fortran
1 * call blacs_gridinfo( ictxt, nprow, npcol, myrow, mycol )
2 * if( nprow.eq.1 ) then
3 * call dtrfx( n, r, G, lld, L, lld )
4 * return
5 * end if
6 * k = 1
7 * nbproc = DESC( MB_ )
```
do while( (k+nbproc).le.n )
c call infog1l( k, nb, nprow, myrow, rsr, lk, pkrow )
c call infog1lh( k, DESC( MB_ ), nprow, myrow, rsr, lk, pkrow )
if( myrow.eq.pkrow ) then
    call dtrfx( nbproc, r, G( lk ), lld, L( lk+k*lld ), lld )
call dlacpy( 'A', nbproc, r, G( lk ), lld, V, nbproc )
broadcast information
call dgebs2d( ictxt, 'C', ' ', nbproc, r, V, nbproc )
else
    call dgebr2d( ictxt, 'C', ' ', nbproc, r, V, nbproc, pkrow, 0 )
end if
kk = k+nbproc
c call infog1l( kk, nb, nprow, myrow, rsr, lk, pkrow )
c call infog1lh( kk, DESC( MB_ ), nprow, myrow, rsr, lk, pkrow )
np = numroc( n-kk+1, nb, myrow, pkrow, nprow )
np = numroch( n-kk+1, DESC( MB_ ), myrow, pkrow, nprow )
call dupdx( np, r, nbproc, G( lk ), lld, V, nbproc, L( lk+k*lld ), lld )
k = k + nbproc
nbproc = DESC( MB_ + pkrow )
end do
c call infog1l( k, nb, nprow, myrow, rsr, lk, pkrow )
c call infog1lh( k, DESC( MB_ ), nprow, myrow, rsr, lk, pkrow )
if( myrow.eq.pkrow ) then
    np = numroc( n-k+1, nb, myrow, pkrow, nprow )
    np = numroch( n-k+1, DESC( MB_ ), myrow, pkrow, nprow )
call dtrfx( np, r, G( lk ), lld, L( lk ), lld )
end if
As it can be seen, this code is essentially the same for the homogeneous case. The BLACS routines are exactly the same, those referent to acquire grid information and the send/receive data. The sequential routines are also the same: \textit{dtrfx} and \textit{dupdx} (lines 6, 20, 39 and 54). However, there exists a difference in the ScaLAPACK tool routines \textit{infog1l} and \textit{numroc}. These must be rewritten in order to work with the new descriptor arrays. The new ones have the same name ending with the character \textit{h}. Each call to the new tool routines appears under the old homogeneous analog one (comment line). The code of \textit{infog1lh} and \textit{numroch} is not much complex than the homogeneous routines as it can be seen in Appendix A. Furthermore, the interface is almost the same except for some arguments that now are integer arrays instead of integer scalars.

Another important difference deals with the different block sizes that each process broadcasts to the rest of processes. The processes that broadcasts a block naturally knows its size (\textit{nbproc}×\textit{r}), where \textit{nbproc} is the size of the actual triangular block that is being computed, but the rests of processes must know which is the block size. All the processes know the size of the actual triangular block (and the size of the communication block) because this information is kept by the descriptor array. That is the reason for our choice as was explained in the introduction to the model. Variable \textit{nbproc} is initiated in line 12 and updated in lines 34, 42 and 43 by all the processes since is a global data.

The main program has the same structure shown in Fig. 11. Now, to build a descriptor array for a one dimensional grid we have implement another heterogeneous routine analog to the ScaLAPACK tool routine \textit{descinit}. This new routine is called from C as

```
   descinit1DH_( desc, &n, &r, nb, &rsrc, &ictxt, &lld, &info );
```

The argument \textit{nb} is an array of the suitable size instead of an integer scalar defining the block size. This routine is sufficient for our algorithm but more work will be needed to implement the equivalent routine for \textit{descinit} for the heterogeneous case.

We have build a performance model with enough detail to the mapping algorithm to map correctly the abstract processors on the suitable real processors. We have not developed a tuning algorithm as the used in the other parallel algorithm for this case, the results have been obtained by manually choosing the best different block sizes according to the different performance of each processor. The number of processes in this case is easy to choose due to it corresponds with the physical processors so the tuning algorithm will be easier than the developed for the other parallel algorithm.

The experimental results obtained with this parallel algorithm are very similar to the ones obtained with the HeHo strategy. The original homogeneous parallel algorithm for the triangular decomposition of a Cauchy–like matrix is more balanced with the increment of the number of row blocks (Fig. 7) but the number of messages increases proportional to this number as well so there exists an optimal value for the block size that allows to avoid overload of messages keeping the workload balanced. With the HoHe parallel algorithm we keep the number of messages down at the same time the workload is balanced thanks to...
6 Conclusions

In this paper we have developed two parallel algorithms to solve a computational kernel very used in a large number of fields like the one shown in Section 2 for the computation of inverse filters in three dimensional sound reproduction systems. Both algorithms have been designed to exploit a heterogeneous NoC’s.

The first one uses the so called HeHo strategy, taking advantage of a work already done for a homogeneous environment of computation or, if not, of the simplicity of programming homogeneous parallel algorithms regarding the complexity of heterogeneous ones. However, a hard work have been made designing a thorough performance model to achieve the best results running a homogeneous application on a heterogeneous environment. Results regarding this performance model as approximation to the expected performance and the speed-up of the parallel algorithm are quite good.

The other parallel algorithm uses the HoHe strategy consisting of an innovative idea that exploits the HeteroMPI, a tool to work not firstly designed with this strategy, to get easiness in the implementation. The main advantage of this idea consists of approximate the homogeneous way of programming to the heterogeneous way without using a new language or different tool from these widely used in the homogeneous NoC’s, that is, ScaLAPACK/BLACS. The main disadvantage regards to the necessity of rewriting the ScaLAPACK routines to accomplish our goal so much work must be done in the future to test the suitability of our proposal for a possible HeteroScaLAPACK.

In addition, in order to implement a parallel application based on calls to ScaLAPACK or HeteroScaLAPACK routines there exists open issues like the one related to how to merge different performance models corresponding to different heterogeneous routines on which a parallel application can be consist of. Although it is always possible to build performance models for any application, it will be convenient to have the availability of a tool capable of merge them in only one without build it from the start.

To finalize, it is important to note that the work developed here represents one of the early experiences developing parallel heterogeneous applications with the recent created HeteroMPI. Through the results obtained under this experience we can say that the technology based on HeteroMPI for programming heterogeneous parallel applications has been validated.
Acknowledgements

This work have been funded by the Spanish Government through the “Secretaría de Estado de Universidades e Investigación del Ministerio de Educación y Cultura” to support a stay of the first author in the Department of Computer Science of the University College of Dublin under the supervision of Dr. A. Lastovetsky. A mention must also be made to the programmer of HeteroMPI, Ravi Reddy, for his help in the installation, learning and support of all the software used to perform this research.

References


REFERENCES


A Code of HeteroScaLAPACK routines

SUBROUTINE INFOG1LH( GINDX, NB, NPROCS, MYROC, ISRCPROC, LINDX, $ ROCSRC )

implicit none

*  -- HeteroScaLAPACK tools routine --
*  Pedro Alonso
*  November 3, 2005
*
*  .. Scalar Arguments ..
 INTEGER GINDX, ISRCPROC, LINDX, MYROC, NPROCS, ROCSRC
 INTEGER NB( NPROCS )
*
* Purpose
* ========
*
* INFOG1LH computes the starting local indexes LINDX corresponding to
* the distributed submatrix starting globally at the entry pointed by
* GINDX. This routine returns the coordinates of the process in the
* grid owning the submatrix entry of global index GINDX: ROCSRC.
* INFOG1LH is a 1-dimensional version of INFOG2L.
*
* Arguments
* ========
*
* GINDX  (global input) INTEGER
* The global starting index of the submatrix.
*
* NB  (global input) INTEGER array ( NPROCS )
* The block size. Each block size belongs to a different processor.
*
* NPROCS  (global input) INTEGER
* The total number of processes over which the distributed
* submatrix is distributed.
*
* MYROC  (local input) INTEGER
* The coordinate of the process calling this routine.
*
* ISRCPROC  (global input) INTEGER
* The coordinate of the process having the first entry of
* the distributed submatrix.
*
* LINDX  (local output) INTEGER
* The local starting indexes of the distributed submatrix.
* * RCSRC (global output) INTEGER
* The coordinate of the process that possesses the first
* row and column of the submatrix.
* *
*====================================================================*
* *
* .. Local Scalars ..
   INTEGER      I, PROC, NPROCS
* ..
* .. Intrinsic Functions ..
   INTRINSIC  MOD
* ..
* .. Executable Statements ..
*
   LINDX = 1
   PROC = ISRCPROC
   I = 1
   DO WHILE( I.LE.GINDX )
      NBP = NB( PROC+1 )
      IF( (I+NBP).LE.GINDX ) THEN
         IF( MYROC.EQ.PROC ) LINDX = LINDX + NBP
      ELSE
         IF( MYROC.EQ.PROC ) LINDX = LINDX + GINDX - I
         RCSRC = PROC
      END IF
      I = I + NBP
      PROC = MOD( PROC+1, NPROCS )
   END DO
*
   RETURN
*
  End of INFOG1LH
*
END
*

 advisory

INTEGER FUNCTION NUMROCH( N, NB, IPROC, ISRCPROC, NPROCS )
implicit none
*
* -- HeteroScalAPACK tools routine --
* Pedro Alonso
* November 3, 2005
*
* .. Scalar Arguments ..
  INTEGER IPROC, ISRCPROC, N, NPROCS
  INTEGER NB( NPROCS )
* ..
* Purpose
* ========
* NUMROCH computes the NUMBER of Rows Or Columns of a distributed
* matrix owned by the process indicated by IPROC.
* * Arguments
* ========
* * N (global input) INTEGER
* The number of rows/columns in distributed matrix.
* * NB (global input) INTEGER array ( NPROCS )
* Block size, size of the blocks the distributed matrix is
* split into. Each block size belongs to a different processor.
* * IPROC (local input) INTEGER
* The coordinate of the process whose local array row or
* column is to be determined.
* * ISRCPROC (global input) INTEGER
* The coordinate of the process that possesses the first
* row or column of the distributed matrix.
* * NPROCS (global input) INTEGER
* The total number processes over which the matrix is
* distributed.
* * NOTE
* ====
* This version only works for pieces of n elements starting at
* the first row of a block as the original ScaLAPACK routine.
* *
* ==========================================================================
*
* .. Local Scalars ..
  INTEGER M, PROC, NBIPROC
* ..
* .. Intrinsic Functions ..
  INTRINSIC MOD, MIN
* ..
*
* .. Executable Statements .. *

M = N
NUMROCH = 0
PROC = ISRCPROC
NBIPROC = NB( IPROC+1 )
DO WHILE( M.GT.0 )
  IF( PROC.EQ.IPROC ) NUMROCH = NUMROCH + MIN( NBIPROC, M )
  M = M - NB( PROC+1 )
  PROC = MOD( PROC + 1, NPROCS )
END DO

* RETURN

* End of NUMROCH

* END