Integration and Development of Routines for the Parallel Solution of Lyapunov Equations by Hammarling's Method [1]

David Guerrero[2], Vicente Hernández[2], José E. Román[2]

June 2001

[1]This document presents research results of the European Community BRITE-EURAM III Thematic Networks Programme NICONET (contract number BRRT-CT97-5040) and is distributed by the Working Group on Software WGS. *WGS secretariat*: Mrs. Ida Tassens, ESAT - Katholieke Universiteit Leuven, Kasteelpark Arenberg 10, 3001-Leuven-Heverlee, BELGIUM. This report is also available by anonymous ftp from *wgs.esat.kuleuven.ac.be* in the directory *pub/WGS/REPORTS/SLWN2001-3.ps.Z*

[2]Departamento de Sistemas Informáticos y Computación (DSIC). Universidad Politécnica de Valencia (UPV). Valencia. Spain. *dguerrer@dsic.upv.es, vhernand@dsic.upv.es, jroman@dsic.upv.es.*

Abstract

This work describes the integration of some routines to solve standard Lyapunov equations by Hammarling's method on parallel platforms. This operation is needed in the parallelisation of the SLICOT routines for reduction of stable systems. Routines from the standard libraries LAPACK, SLICOT, PBLAS and ScaLAPACK have been used whenever possible. However, it has been necessary to develop some new routines. Experimental results are shown, which have been obtained using a cluster of PC's.

Contents

1 Introduction

The problem of solving the standard Lyapunov equation appears frequently in many problems related to the control theory. In particular, it is used as a basis in the main algorithms used to perform model reduction of stable systems. Developing a parallel solver for this equation is a first step to parallelise SLICOT routines for reduction of stable systems, such as AB09AD, AB09BD and AB09CD.

There are several implementations to solve this equation in the sequential case [8]. However, the parallel case has not been explored in such an extended way. This document shows the work carried out to obtain a parallel solver for the standard Lyapunov equation following the method due to Hammarling [7].

Some of the needed routines are available in the standard parallel library ScaLAPACK [3]. However, some of them are not yet. Thus, some new parallel routines have been developed to allow for the complete process of solving standard Lyapunov equations in parallel.

The structure of the next sections is as follows. First, the form of the Lyapunov equation is presented. Then, the Hammarling method for solving it is explained. After this, a description of the parallel routines needed to perform the task is done, focusing on the newly developed ones. Finally, some experimental results are shown for the execution of the solver in a cluster of PC's interconnected by a Gigabit Ethernet network.

2 Hammarling's method for Lyapunov equations

2.1 Standard Lyapunov equation

The standard Lyapunov equation is presented in two forms, one for continuous-time systems

$$A^T X + XA = -Y \tag{1}$$

and one for discrete-time systems

$$A^T XA - X = -Y. \tag{2}$$

Here A and Y are real square matrices of size n. Matrix Y is symmetric, as well as the solution matrix X when unique.

The most typical method used to solve these equations is due to Bartels and Stewart [2]. It is based on transforming the equation to a reduced form, solve this reduced form and then transform back the solution.

2.2 Hammarling's method

Another method due to Hammarling [7] is an alternative for solving Lyapunov equations when their right-hand side is positive semidefinite and matrix A is stable. In these cases, the right-hand side of the equation is usually in the form of a product of a matrix B of size $m \times n$ by its transposed matrix, and the Cholesky factor U of the solution matrix X is the desired output. The above equations become

$$A^T U^T U + U^T U A = -B^T B, \tag{3}$$

for the continuous case, or

$$A^T U^T U A - U^T U = -B^T B, \tag{4}$$

for the discrete one.

This method allows to obtain the Cholesky factor of the solution directly without needing to perform the product in the right-hand side of the equation explicitly.

It is similar to Bartels-Stewart method in that both of them work by transforming the equation to a reduced form, then solving this reduced equation and later obtaining the solution to the original equation by a back transformation.

These steps are going to be described in detail for the continuous case of Hammarling's method.

2.2.1 Transformation to reduced form

The transformation to reduced form is performed to obtain the equation

$$A_s^T X_s + X_s A_s = -B_s^T B_s, \tag{5}$$

where B_s is an upper triangular matrix of size $n \times n$, and A_s is an upper quasi-triangular matrix of the same dimensions.

This form of the equation is obtained by reducing matrix A to the real Schur form A_s, via computing the orthogonal matrix Z which verifies

$$A_s = Z^T A Z, \tag{6}$$

being A_s an upper quasi-triangular matrix, that is, upper triangular with the exception of some nonzero elements in the first subdiagonal. A_s can be seen as an upper block triangular matrix, whose diagonal is formed by 1×1 or 2×2 blocks corresponding to real or complex eigenvalues of A, respectively.

In this transformation to reduced form (5), matrix B_s is obtained as the upper triangular matrix R from the QR factorisation of the product BZ.

2.2.2 Solution of the reduced equation

To solve equation (5) for U_s, Cholesky factor of the solution ($X_s = U_s^T U_s$), the involved matrices A_s, B_s and U_s are partitioned in a way that yields a 2 by 2 or 1 by 1 Lyapunov equation and two other equations. This small Lyapunov equation is solved as a linear system of equations (by Kronecker products). Then the other two equations are updated. One of them is solved as a reduced Sylvester equation in which involved matrices are upper quasi-triangular. The other equation is transformed by a QR decomposition and matrix products to a reduced Lyapunov equation which can be solved by this same procedure.

Note that the size of the reduced Lyapunov equation to solve is reduced every time this procedure is applied by 1 or 2 depending on the partitioning used, which also depends on the form of matrix A_s. Thus, this is a finite process.

The solution of this reduced equation in parallel has been treated previously for the generalised case [6] and for the standard case [5].

2.2.3 Back transformation

Once the solution of the reduced equation has been computed, another transformation is required to obtain the solution of the original equation.

In Hammarling's method, this transformation consists in obtaining the QR factorisation of the product of matrix U_s, solution of the reduced equation, and the transpose of matrix Z, coming from the transformation to real Schur form

$$U_s Z^T = Q_U U. \tag{7}$$

With this transformation, the upper triangular matrix U, Cholesky factor of the solution X of equation (1), is obtained.

3 Parallel implementation

The procedure to obtain a parallel code for the complete solution of Lyapunov equations by the Hammarling's method is based on using available parallel routines for each of the basic operations performed in every step. However, some of those operations had not still been implemented in parallel. New parallel routines have been developed for them.

Figure 1 shows the call tree for the main routines used in the parallel solution of Lyapunov equations. In this figure, the new routines developed by the authors are shown in bold face. These routines are explained next.

3.1 PDLPHM

This is the main routine to be called for solving Lyapunov equations by Hammarling's method.

It allows to solve both continuous and discrete time Lyapunov equations, as well as their transpose versions. It also has an output scale factor set to avoid a possible overflow in the solution.

The Schur factorisation needed for the step of reducing the equation may have been done outside the routine, thus allowing to avoid this operation when calling **PDLPHM** multiple times to solve several equations with the same A matrix. In this case, the first call may be specified as not having performed the Schur decomposition, which will then be performed by the routine. Successive calls will make use of the Schur decomposition previously computed.

$$
\textbf{PDLPHM}\left\{\begin{array}{l}
\textbf{PDGEES}\left\{\begin{array}{l}
\text{PDGEHRD}\\
\textbf{PDORGHR}\,\{\text{PDORGQR}\\
\text{PDLAHQR}\\
\textbf{PDLANV}\,\{\text{DLANV2}
\end{array}\right.\\
\textbf{PDGEMM2}\,\{\text{DGEMM}\\
\text{PDGEQRF}\\
\text{PDGERQF}\\
\textbf{PDSHIFT}\\
\textbf{PDDIAGZ}\,\{\text{DSCAL}\\
\textbf{PSB03OT}\left\{\begin{array}{l}
\text{SB03OT}\\
\textbf{PDTRSCAL}\,\{\text{DSCAL}\\
\textbf{PSB03OR}\,\{\text{SB03OR}\\
\textbf{PMB04OD}\left\{\begin{array}{l}\text{DLARFG}\\ \text{MB04OY}\end{array}\right.\\
\textbf{PMB04ND}\left\{\begin{array}{l}\text{DLARFG}\\ \text{MB04NY}\end{array}\right.
\end{array}\right.\\
\textbf{PDTRMM2}\left\{\begin{array}{l}\text{DTRMM}\\ \text{DGEMM}\end{array}\right.
\end{array}\right.
$$

Figure 1: Call tree of the main routines.

This routine can be considered a parallel version for the standard equation of the similar routine for the generalised equation in the sequential case DGLPHM [8]. It is almost the same as the sequential routine except for calling parallel versions of the needed operations.

It also can be seen as a parallel version of the SLICOT routine SB03OD, since it offers the same functionality.

The routine performs the reduction of the equation, solution of the reduced equation and back transformation of the solution.

The reduction of the equation is done by calling **PDGEES** to obtain the Schur form of matrix A thus obtaining the reduced left hand side part of the equation. The right-hand side is obtained by multiplying matrix B with the orthogonal matrix returned by **PDGEES** using **PDGEMM2**, and later performing a QR factorisation (RQ in the transpose case) with the ScaLAPACK routine PDGEQRF (PDGERQF). Reduced right-hand side is obtained then by scaling this matrix in order to make all its diagonal entries non-negative, with the **PDDIAGZ** routine. In the transpose case with a non-square B matrix, it is necessary to make a redistribution of the matrix after the call to PDGERQF routine which is done in the **PDSHIFT** routine.

The solution of the reduced equation is computed by using the routine **PSB03OT**[5].

Transformation of the reduced equation solution to the desired solution is accomplished by multiplying it by the orthogonal matrix returned by **PDGEES**, which is done with the **PDTRMM2** routine, and then obtaining the QR factorisation (RQ in the transpose case) of this product with the ScaLAPACK routine PDGEQRF (PDGERQF). Once again, entries on the main diagonal are made non-negative using the routine **PDDIAGZ**.

3.2 PDGEES

This routine obtains the real Schur form of a generic matrix. The resulting Schur form is in standard form, that is, the 2×2 blocks which may appear in the main block diagonal are in the form

$$\begin{bmatrix} a & b \\ c & a \end{bmatrix}$$

where $b * c < 0$, being $a \pm \sqrt{bc}$ the corresponding eigenvalues of that block.

The routine is based on the sequential LAPACK[1] routine DGEES. The transformation is performed in two steps, first the matrix is transformed to upper Hessenberg form, and later this Hessenberg matrix is transformed to Schur form.

The transformation to upper Hessenberg form is performed by the ScaLAPACK routine PDGEHRD. This routine returns the orthogonal transformation matrix used in the form of reflectors and τ values (LAPACK and ScaLAPACK way of storing orthogonal matrices). This has to be transformed into an standard matrix by the routine **PDORGHR**.

The Hessenberg to Schur transformation is made by using the ScaLAPACK routine PDLAHQR [3]. However, this routine currently has several restrictions. It looks to be a preliminary version of what it should be. In particular, it does not return the 2×2 blocks in the mentioned standard form, whereas the equivalent LAPACK routine DLAHQR does. Thus, the routine **PDLANV** has to be called in order to standardise them.

3.3 PDORGHR

This routine converts the input orthogonal matrix stored as reflectors and τ values to a regular matrix.

It is a parallel version of the LAPACK routine DORGHR. It basically fills to zero some portions of the matrix, shifts it and then calls the ScaLAPACK routine PDORGQR which makes the operation.

3.4 PDLANV

This routine has been required since the routine PDLAHQR currently available in ScaLAPACK does not return the 2×2 blocks of the real Schur form matrix in standard form. This routine converts these blocks into standard form and it also accumulates in the appropriate way the used transformations in the orthogonal matrix which represents the operations performed to transform the original matrix to Schur form. This is done by calling LAPACK routine DLANV2 for each block and then updating the corresponding rows and columns of the Schur form matrix and the orthogonal matrix accordingly.

First the three main diagonals of the matrix in non-standard Schur form are broadcasted by rows and columns of the processor grid in a way that every processor has the corresponding elements of those three diagonals for its rows and columns. In this way, each processor can call DLANV2 for each 2×2 block and update its matrices, except for the cases of 2×2 blocks owned by several processors. For these blocks, extra communications are performed.

3.5 PDGEMM2

This routine performs the operation $B \leftarrow \mathrm{op}(A) \times B$ or $B \leftarrow B \times \mathrm{op}(A)$, where $\mathrm{op}(A)$ is A or A^T, i.e. it computes the product of two general matrices overwriting one of them with the result.

This operation could have been done with the PBLAS routine PDGEMM, which permits to compute the product of two matrices storing the result in another one, and later copying it to the result, but space for an extra matrix would be necessary in this case.

In the sequential case, this operation is performed row by row (or column by column for the transpose equation case) by copying each row (column) to an extra row (column) space and using the BLAS routine DGEMV to perform matrix by vector multiplications. This same way of proceeding can be used in the parallel version, but the use of the PBLAS routine PDGEMV n times (for an $n \times n$ matrix) implies n times the cost of performing one simple distributed matrix by vector product.

A first attempt to improve this is by using a block-oriented version. A product of the full matrix by a block of rows (or columns) is made each time. This is done by copying the corresponding block to an auxiliary space and then using the PBLAS routine PDGEMM. In this way a level-3 BLAS routine is used.

Following that idea but trying to minimise the required communications, a new strategy has been implemented to perform each of the four possible products: $B \leftarrow A \times B$, $B \leftarrow A^T \times B$, $B \leftarrow B \times A$ and $B \leftarrow B \times A^T$.

Figure 2 shows the communications required to compute the first column block of the product $A \times B$. Since matrix B is going to be overwritten with the product, the operation is performed by blocks of columns. The scheme shown in figure 2 is repeated for each column block of matrix B. Note that each letter in the figure represents a block of the corresponding matrix and not a single element.

Similar schemes are used for the other three variants of the operation, but with different communication pattern which is adapted for each of them.

3.6 PDSHIFT

This routine performs a horizontal displacement of a matrix. It is needed in the transpose case of the equation if the original matrix B of the right hand side is not square.

3.7 PDDIAGZ

This routine is used several times in the code to make the elements in the main diagonal of a matrix non-negative by scaling with minus one the rows (or columns) corresponding to negative

$$
\begin{array}{ccc}
\begin{array}{cccc}
b_1 & \times & \dots & \times \\
b_2 & \times & \dots & \times \\
\vdots & \vdots & \ddots & \vdots \\
b_n & \times & \times & \times
\end{array}
&
\begin{array}{cccc}
b_1 & & & \\
& b_2 & & \\
& & \ddots & \\
& & & b_n
\end{array}
&
\begin{array}{cccc}
b_1 & b_2 & \dots & b_n \\
b_1 & b_2 & \dots & b_n \\
\vdots & \vdots & \ddots & \vdots \\
b_1 & b_2 & \dots & b_n
\end{array}
\\
\text{Original distribution} & \text{Redistribute} & \text{Broadcast by columns}
\end{array}
$$

$$
\begin{array}{cc}
\begin{array}{cccc}
a*b_1 & a*b_2 & \dots & a*b_n \\
a*b_1 & a*b_2 & \dots & a*b_n \\
\vdots & \vdots & \ddots & \vdots \\
a*b_1 & a*b_2 & \dots & a*b_n
\end{array}
&
\begin{array}{cccc}
a*b_1 + a*b_2 + \ldots + a*b_n & \times & \dots & \times \\
a*b_1 + a*b_2 + \ldots + a*b_n & \times & \dots & \times \\
\vdots & \vdots & \ddots & \vdots \\
a*b_1 + a*b_2 + \ldots + a*b_n & \times & \dots & \times
\end{array}
\\
\text{Multiply by } A & \text{Reduce to a column}
\end{array}
$$

Figure 2: Communications to compute the first column block of $A \times B$.

diagonal elements. Trapezoidal matrices are allowed.

The way of proceeding is the following: first all processors have to know the values of the main diagonal entries of rows (or columns) from which they have elements. Then, all of them perform in parallel the multiplication by minus one of the rows (or columns) whose associated main diagonal element is negative.

3.8 PSB03OT

This routine is a parallel version of the SLICOT routine SB03OT, which is used to solve a *reduced* Lyapunov equation obtaining the Cholesky factor of the solution.

It has been implemented as a blocked version which calls SB03OT to solve the current block of the equation and then updates the rest of the equation in the same way as the SLICOT routine does but with parallel versions of the involved routines: **PSB03OR**, **PMB04OD** and **PMB04ND**.

One important problem to solve here was the case when a block of the equation cuts a 2×2 block, associated with complex eigenvalues, in the main diagonal of the involved matrix. This case is treated in the implementation, which produces extra communications to send the next row/column of the matrices and so treating it as a whole. The 2×2 blocks are always solved together, although this implies extra communications.

Moreover, in order to maintain the functionality of the sequential routine and so allowing an extra output parameter SCALE which may be set to avoid overflow in the solution, a new routine has been developed to scale triangular matrices: **PDTRSCAL**.

More information about this routine can be found in [5]

3.9 PDTRSCAL

This routine allows to scale a triangular (or trapezoidal) distributed matrix by a value.

Each processor scales its corresponding portion of the matrix.

3.10 PSB03OR

PSB03OR is a parallel version of the SLICOT routine SB03OR.

In the same way that SB03OR is for SB03OT, **PSB03OR** is a *service routine* for **PSB03OT**, that is a routine developed to complement the need of solving a type of Sylvester equation in **PSB03OT**.

PSB03OR allows to solve in parallel Sylvester equations whose matrices are in Schur form. SB03OR requires one of the matrices to be of size at most 2. This was enough for the sequential implementation in SB03OT. However, in the parallel case **PSB03OT** needs to solve Sylvester equations in which one of the matrices is of any size but the other is not restricted only to a size of at most 2. This matrix can be of a greater size, limited by the block size in each moment. It is owned completely by one processor and previously to call to this routine it must be replicated in all the processors in the adequate context (in a column or row of processors depending on whether it is the transpose version or not).

PSB03OR is a parallel version of a blocked SB03OR routine. Therefore, it calls SB03OR to solve a Sylvester equation completely owned by a processor and then updates the rest to continue solving in the next iteration. Note that since SB03OR requires one of the involved matrices to be of size at most 2, the solution of every Sylvester equation completely owned by a processor requires, in general, more than one call to the sequential routine. A loop is made to solve this local equation which consists in calling SB03OR several times and updating the equation accordingly.

Similarly to what occurred with the **PSB03OT** routine, a problem may happen if a 2×2 block of the matrix involved in the Sylvester equation is owned by more than one processor. In these cases communications have to be made to guarantee that these blocks are not cut. The elements on this blocks are solved in a unique Sylvester equation whose size is incremented by one. After this, extra communications are also required to return the solution to its owner processor(s).

3.11 PMB04OD and PMB04ND

These routines are parallel versions of the SLICOT routines MB04OD and MB04ND. However, they are light versions in the sense that the parallel versions do not have all the functionality offered by sequential ones. They have only the functionality required by **PSB03OT**.

These routines perform QR (**PMB04OD**) or RQ (**PMB04ND**) factorisations and apply the corresponding orthogonal transformations to a specified matrix.

The way of proceeding is by computing successively appropriate reflectors by calling the LAPACK routine DLARFG and then applying them in parallel to the matrices with the SLICOT routines MB04OY and MB04NY.

3.12 PDTRMM2

This routine performs a product of a general matrix with a triangular matrix storing the result in the triangular matrix, which must have enough space to store it (the result will not be triangular in general). It is similar to the PBLAS routine PDTRMM, but PDTRMM stores the result in the general matrix. For this reason, PDTRMM would be an alternative if using an extra space to copy one of the matrices.

In the sequential implementation, this operation is performed by a loop with several matrix by vector products. In parallel, a blocked algorithm is better.

The implementation of this routine is very similar to that of **PDGEMM2** routine. The main differences are that the triangular form of one of the matrices is now exploited both in the communications and in the operations to perform with it. However, the scheme of the process is the same. An example was shown in figure 2.

4 Experimental results

4.1 Parallel platform

The parallel platform used to test the routines is a cluster of PC's. Each node in the cluster is a biprocessor computer with 2 Pentium III processors at 866 MHz, 512 Mb of RAM and Redhat Linux operating system. Each node also has three network interfaces: two Fast Ethernet interfaces integrated in the motherboard and one Gigabit Ethernet card.

The implemented routines have been tested with up to 5 nodes. However, each node has been treated as a single processor machine. No exploitation of the biprocessor feature has been done.

4.2 Problem to solve

The Lyapunov equation used to test the routines is generated in a way similar to that explained in [8], but specialised for the standard case and generating a square matrix B (in the reference B was a transposed vector).

The matrix A of size $n \times n$, $n = 3q$, is generated as

$$A = W_n^{-1}\operatorname{diag}(A_1, \ldots, A_q)W_n, A_i = \begin{pmatrix} s_i & 0 & 0 \\ 0 & t_i & t_i \\ 0 & -t_i & t_i \end{pmatrix}$$

where W_n is a matrix of size $n \times n$ whose elements are all one except those of the main diagonal which are zero. The parameters s_i and t_i determine the eigenvalues of the generated matrix. They are chosen as $s_i = t_i = t^i$ in the continuous case and $s_i = 1 - 1/t^i, t_i = -\sqrt{2}s_i/2$ in the discrete case.

The matrix B of size $n \times n$ is generated as the symmetric definite matrix

$$B = \begin{pmatrix} n & n-1 & \dots & 1 \\ n-1 & n & \dots & 2 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & \dots & n \end{pmatrix}.$$

Two different problems have been generated to test the parallel routines. Both of them belong to the continuous case and are generated as mentioned above with values 1.0 and 1.01 for parameter t.

In the problem with $t = 1.0$, the eigenvalues of matrix A are 1 and $1 \pm i$, all of them with multiplicity q. This case is more intended to test the portion of the parallel code in charge of solving the reduced equation and the part which transforms the right hand side of the equation and the final solution. These steps do not depend significantly on the values of the involved matrices. However, the **PDGEES** routine execution time depends strongly on the eigenvalues of the matrix. Therefore, a matrix with more different eigenvalues has been chosen as the second test case. This matrix has several different eigenvalues, whose value vary from t^1 until $t^{n/3}$ being n the size of the problem. Since the problem size used to measure times is $n = 2001$ (a multiple of 3 for commodity), a value of $t = 1.01$ has been chosen in order to get a difference among eigenvalues lower than $1.01^{2001/3} \simeq 762.70$. The desired effect was to obtain a difference lower than the size of the matrix, since a value of $t = 1.02$ produces eigenvalues of order 544897.19 for a matrix of order 2001.

4.3 Timing results

Previously to proceed with the parallel executions, a block size has been chosen by executing the parallel algorithm on a single node. Figures 3 and 4 show execution times (in seconds) of this sequential runs in order to choose a correct value for the block size.

For the $t = 1.0$ problem a block size of 24 has been chosen to test the parallel routines, which seems to be the best in the plot (see figure 3).

However, the $t = 1.01$ case has more variability, maybe due to a larger size of the problem or the fact of having more different eigenvalues. Although the best times have been obtained for a block size of 8, the finally chosen value was 16. The results for this block size are almost equal in the sequential case, but a larger block size will reduce the number of messages in the parallel algorithm.

Thus, parallel executions have been done for problems of size 2001 with $t = 1.0$, block size 24, and $t = 1.01$, block size 16. Each of these cases has been executed both using the Fast Ethernet network and the Gigabit Ethernet network.

The times obtained in seconds can be seen in figures 5 and 6. As expected, the times using the Gigabit Ethernet network are lower than those of the Fast Ethernet network. However, differences are not as big as they may be expected. The reason for this can be that the developed algorithm does not have a large communications/computations ratio. Computation is still the most costly part. The algorithm seems to use a sufficiently coarse grain, since a better network does not increase performance too much.

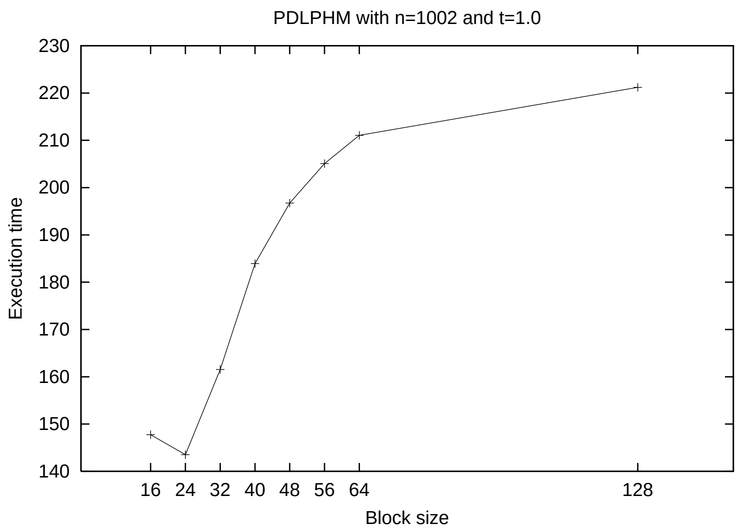

Figure 3: Execution times in one node with different block sizes for t=1.0.

Speedups and efficiencies are shown in tables 1 and 2. The case with $t = 1.0$ has shown very good performance. Efficiencies of 90% up to 4 processors for a problem size of 2001 can be considered good results for this kind of problems (dense matrices).

Sp	P=1	P=2	P=3	P=4	P=5
n=1002	1	1.51	1.92	2.07	2.41
n=2001	1	1.9	2.98	3.58	3.88
E	**P=1**	**P=2**	**P=3**	**P=4**	**P=5**
n=1002	1	0.76	0.64	0.52	0.48
n=2001	1	0.95	0.99	0.90	0.78

Table 1: Speedups and efficiencies for t=1.0.

However, the problem with $t = 1.01$ has much worse results. This implies that the problem of computing the Schur form, whose cost is the one which depends most on the values of the initial problem, has better parallelism for a simpler matrix.

This fact can be seen in the next figures, which represent the execution times of the most important tasks performed in the process, as shown in figure 1.

Figures 7 and 8 show the execution times of the main routines performed in the overall process of solving the Lyapunov equation, these are the main routines called in **PDLPHM**.

Figures 9 and 10 show the execution times of the more important routines used in the Schur factorisation computed in **PDGEES**.

As it has been previously predicted, all the routines, except those which involve directly the Schur factorisation step (**PDGEES**), have similar behaviour for both the problem with $t = 1.0$

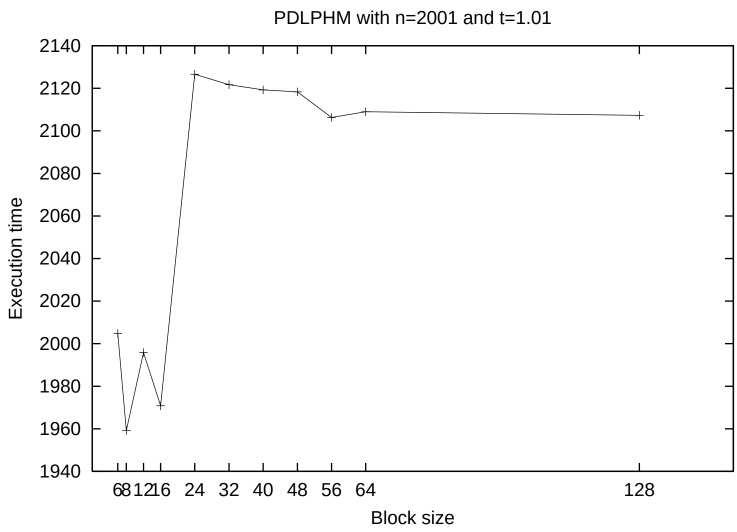

Figure 4: Execution times in one node with different block sizes for t=1.01.

n=2001	P=1	P=2	P=3	P=4	P=5
Sp	1	1.46	2.04	1.58	2.49
E	1	0.73	0.68	0.40	0.50

Table 2: Speedups and efficiencies for t=1.01.

and the one with $t = 1.01$. All of them have the typical form desirable for a parallel algorithm, which is a curve which decreases as the number of processors grows.

However, the graphics associated to the Schur factorisation process show an irregular behaviour. The time sometimes grow and sometimes is reduced as the number of processors grows. When looking at figures 9 and 10, it can be observed that this is due to the ScaLAPACK routine PDLAHQR. It seems not to have a good parallel performance. This can be due to the complexity of performing the Schur decomposition in parallel.

Finally, it must be noted that graphics representing execution times of different routines may not be the optimal cases for those routines. The block size has been chosen trying to obtain the optimal time results for the whole operation of solving the Lyapunov equation. Thus, the times obtained for a particular routine may not show the optimal execution time of that routine.

5 Conclusions

In this report, the integration of routines to obtain a parallel algorithm to solve standard Lyapunov equations by the Hammarling's method has been presented.

The first thing that must be noted is that only a part of the routines needed to perform

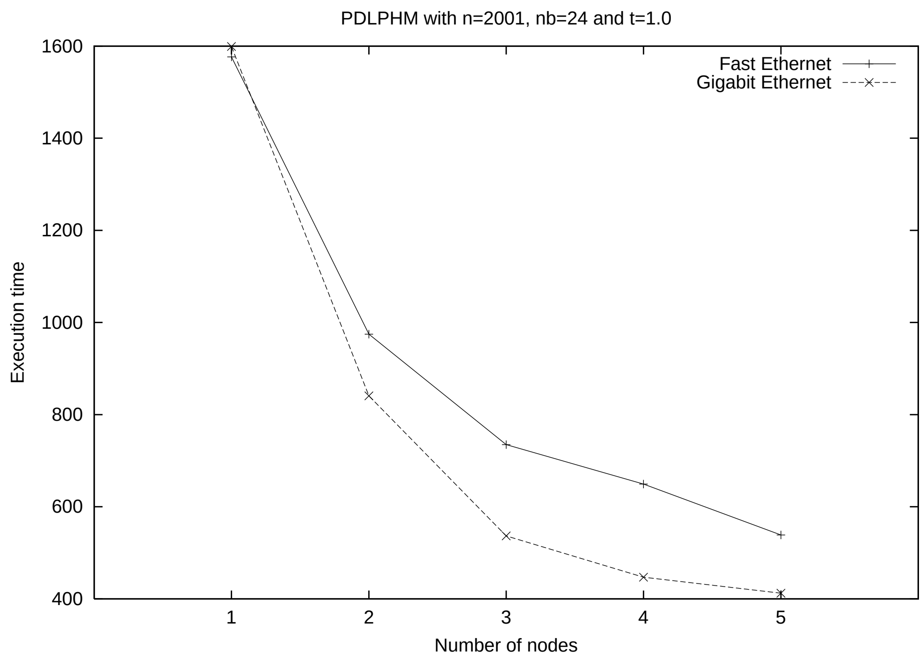

Figure 5: Parallel execution times for t=1.0.

this task was implemented in ScaLAPACK, thus being necessary to make implementations for the rest of the operations. The new routines have been necessary mainly for the tasks of Schur factorisation (**PDGEES** routine is not completed in ScaLAPACK, yet the major part of its required routines are), the step of solving the reduced equation (explained in another report [5]), and some minor routines needed in the transformations (to compute the product of two matrices storing it in one of them).

Once all the necessary routines were available, the integration of calls among them has been developed. The result is a main routine called **PDLPHM** which allows to solve standard Lyapunov equations in parallel, both continuous and discrete versions, both in transpose or non-transpose form. This new routine and all the newly developed ones follow the classical syntax and calling convention of ScaLAPACK, thus allowing any user of ScaLAPACK to work easily with them.

The parallel implementation has been tested with some problems in order to see the performance it can achieve. This performance has shown to be rather good. Only a part of the algorithm seems not to be well parallelised. This is the part in charge of the Schur factorisation, in particular the ScaLAPACK routine PDLAHQR. However, it may be due to the complexity to parallelise this operation. Note that all the other ScaLAPACK routines used (mainly PDGEHRD, PDORGQR, PDGEQRF) show good parallel results.

As a conclusion, it can be said that a good parallel algorithm to solve several forms of the standard Lyapunov equation has been developed making use of standard libraries, such as BLAS, LAPACK, SLICOT, BLACS[4] and ScaLAPACK. Although the implemented subroutines work

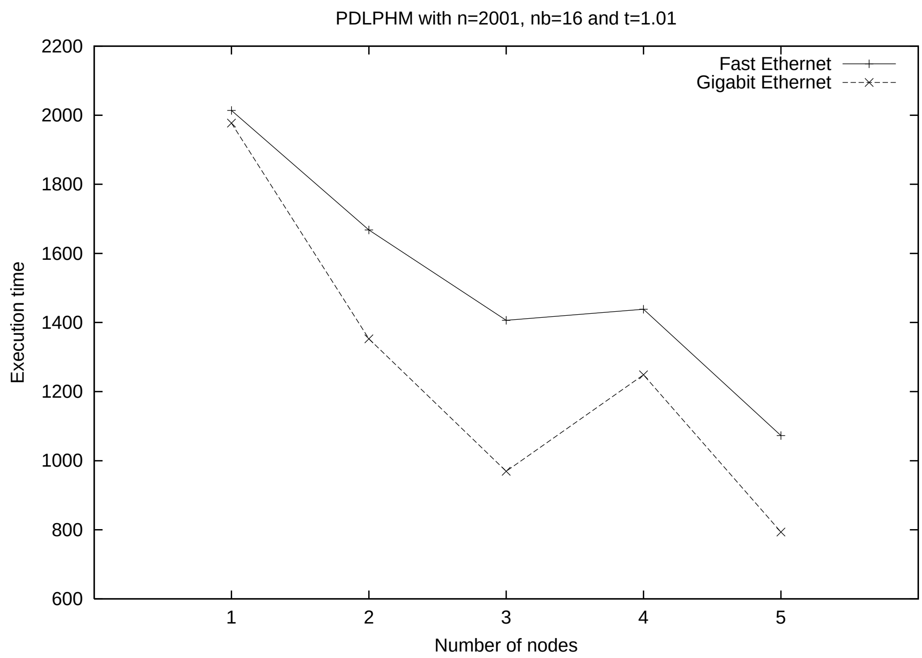

Figure 6: Parallel execution times for t=1.01.

perfectly with the two presented test cases, the authors plan to carry out further testing and experimentation with a more complete test battery.

Moreover, now that a parallel routine to solve standard Lyapunov equations is available, it is possible to deal with the problem of parallelising the main SLICOT routines for the reduction of stable systems. This work is now being performed by the authors.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorenson. *LAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[2] R. H. Bartels and G. W. Stewart. Solution of the equation $AX + XB = C$. *Comm. ACM*, 15:820–826, 1972.

[3] Jaeyoung Choi, Jack J. Dongarra, Roldan Pozo, and David W. Walker. ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers. Mathematical Sciences Section, Oak Ridge National Laboratory, 1992.

[4] Jack J. Dongarra and R. Clint Whaley. LAPACK working note 94: A user's guide to the BLACS v1.0. Technical Report UT-CS-95-281, Department of Computer Science, University of Tennessee, March 1995.

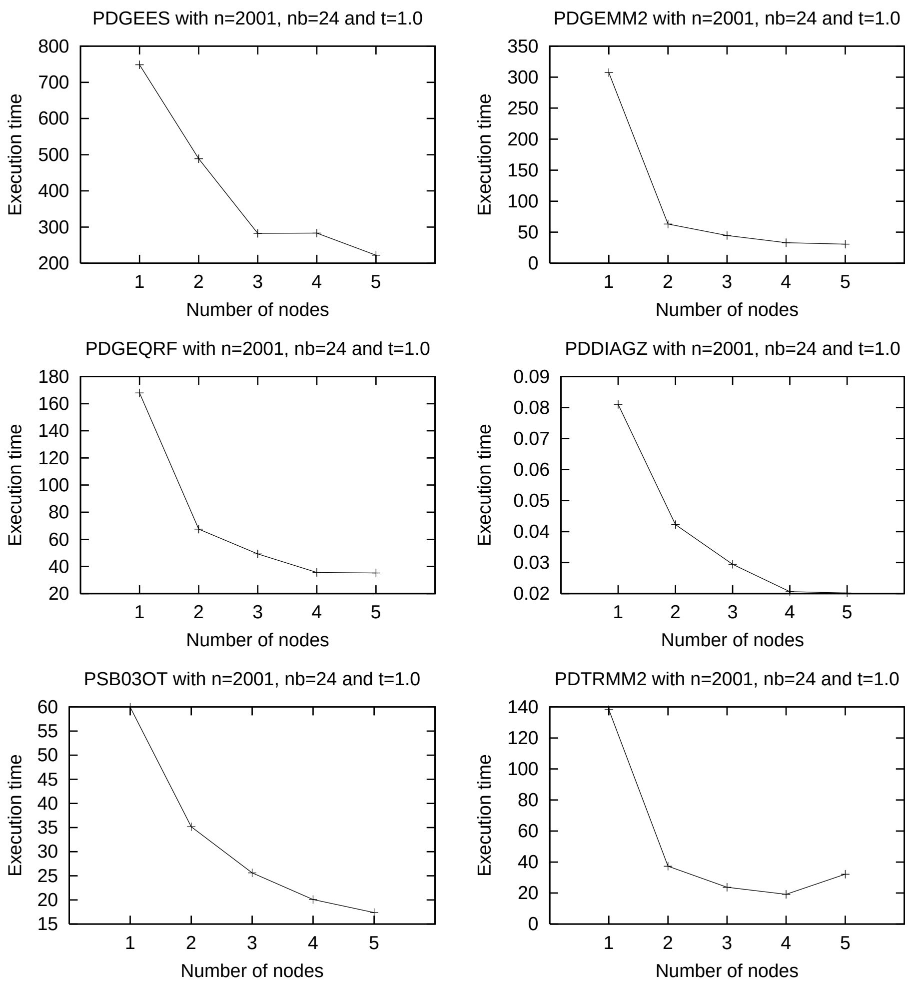

Figure 7: Parallel execution times of the main routines in **PDLPHM** for t=1.0.

[5] David Guerrero, Vicente Hernández, and José E. Román. Parallel Solution of the Standard Lyapunov Equation by Hammarling's Method. Technical report, Third NICONET Workshop on Numerical Control Software, Louvain-la-Neuve (Belgium), January 2001.

[6] David Guerrero, Vicente Hernández, José E. Román, and Antonio M. Vidal. Parallel Algorithms for the Cholesky Factor of Generalized Lyapunov Equations. In *5th IFAC Workshop on Algorithms and Architectures for Real-Time Control*, pages 237–242, Cancun, Mexico, April 1998.

[7] S. J. Hammarling. Numerical solution of the stable, non-negative definite Lyapunov equation. *IMA J. of Numerical Analysis*, 2:303–323, 1982.

[8] Thilo Penzl. Numerical solution of generalized Lyapunov equations. Technical Report SFB393/96-02, Numerische Simulation auf massiv parallelen Rechnern, Technische Univer-

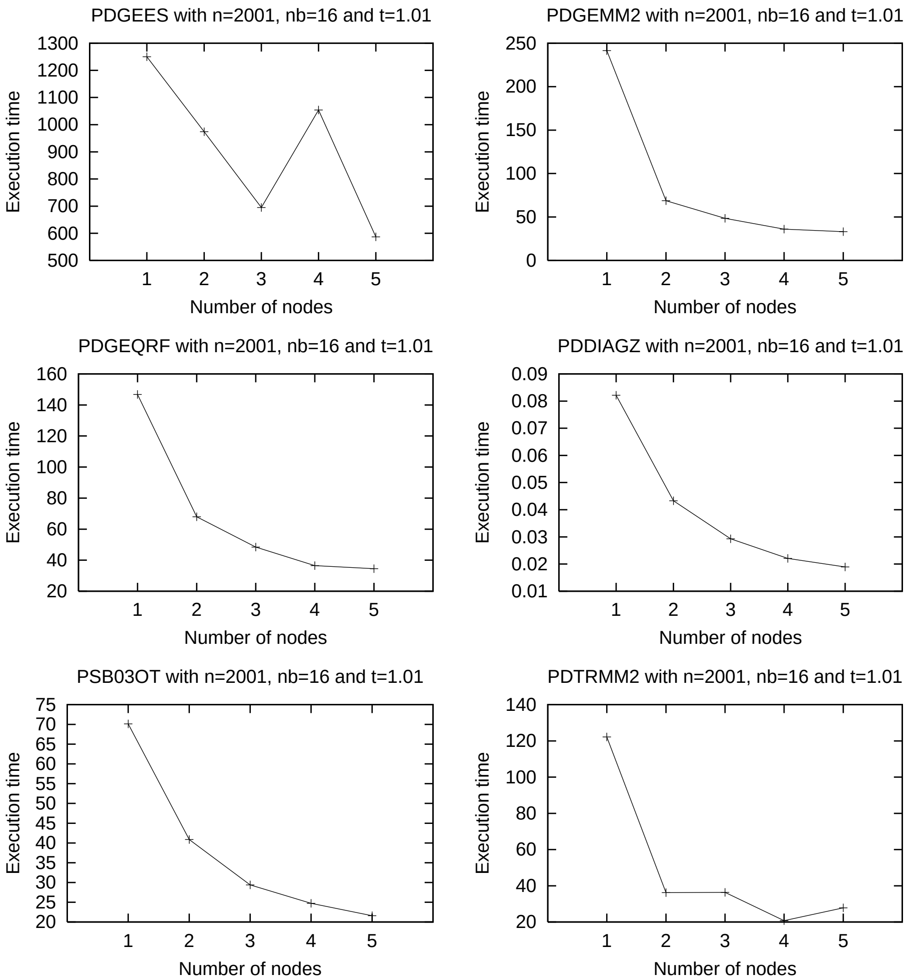

Figure 8: Parallel execution times of the main routines in **PDLPHM** for t=1.01.

sität Chemnitz-Zwickau, May 1996.

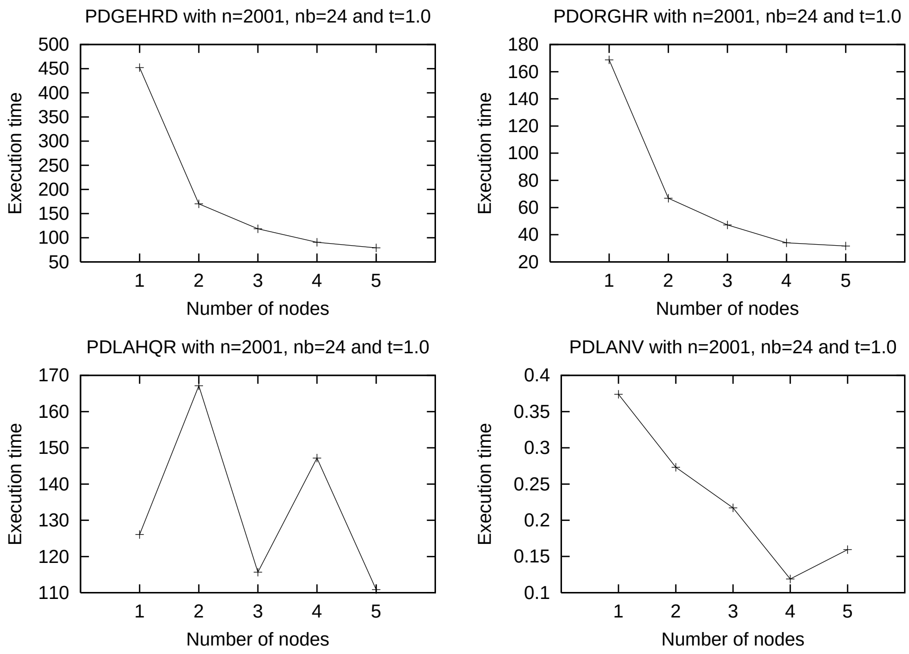

Figure 9: Parallel execution times of the main routines in **PDGEES** for t=1.0.

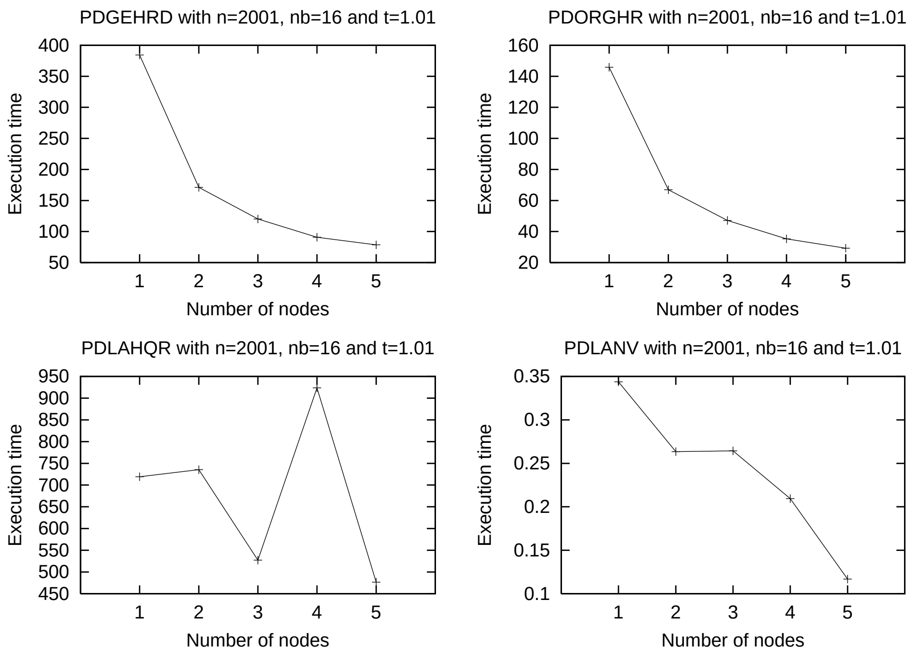

Figure 10: Parallel execution times of the main routines in **PDGEES** for t=1.01.