629.8(S)
X

# Demonstration of the MATLAB Parallel Processing Gateway

A. J. Chipperfield and P. J. Fleming

Department of Automatic Control and Systems Engineering
University of Sheffield
PO Box 600
Mappin Street
Sheffield S1 4DU

e-mail: A.Chipperfield@shef.ac.uk
P.Fleming@shef.ac.uk

September 29th 1993

*ACSE Research Report No. 487*

**Abstract:** This report provides a tutorial intorduction to the MATLAB parallel processing gateway developed at Sheffield. It is intended that this document be read in conjuction with an interactive MATLAB tutorial script-file, ppdemo. Together, the tutorial script and this document take the reader through the processes of configuring, booting and using parallel routines from MATLAB through the use of a multiobjective optimization example.

# Demonstration of the MATLAB Parallel Processing Gateway

Andrew Chipperfield and Peter Fleming

Department of Automatic Control and Systems Engineering
University of Sheffield
PO Box 600
Mappin Street
Sheffield S1 4DU

## 1. Introduction

This document is intended to provide a tutorial introduction to the parallel processing gateway developed at Sheffield and demonstrate how routines running on a parallel platform can be called from the MATLAB [1] address space. A script-file, **ppdemo.m**, accompanies this document and takes the reader through the process of configuring, booting and using parallel routines from MATLAB through the use of a multiobjective optimization problem.

## 2. The Parallel CACSD Environment

The parallel CACSD environment, shown in Fig, 1, is based around the commercially available MATLAB software package with a number of extensions to support parallel processing, collectively called the "Parallel Processing Gateway". The parallel processing gateway is described in full detail in ACSE Research Report No. 486 [2], and [3], and is described here for completeness.

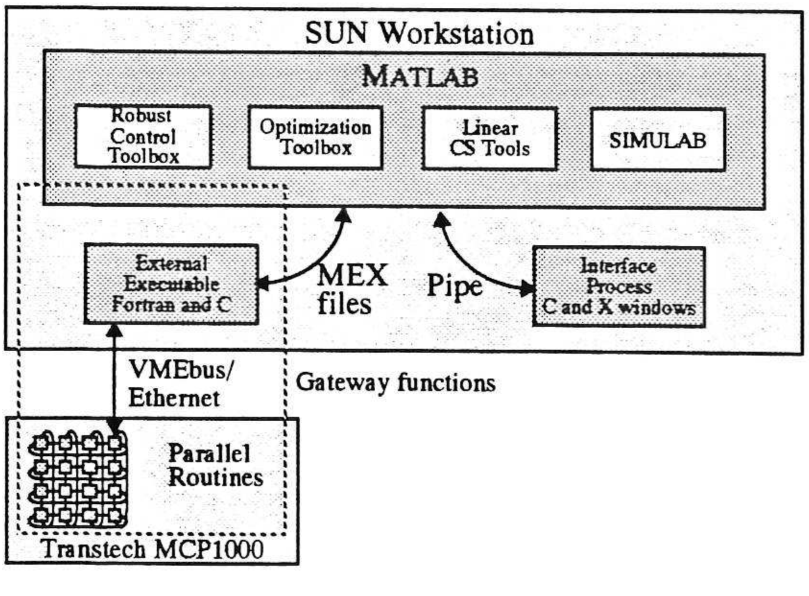

Fig. 1: The Parallel CACSD Environment

The parallel processing gateway functions allow the user to configure and boot processes on the parallel platform and access the parallel routines transparently from within MATLAB. These gateway functions act as a buffer between the different formats employed by MATLAB and the GENESYS operating system running on the transputer platform. As such, routines running on the parallel platform appear functionally equivalent to the same routine coded as an m-file or mex-file and can be called directly from the MATLAB command line, from within an m-file or mex-file and directly from other routines running on the platform. A routine running on the parallel platform is called from MATLAB using the standard form, i.e.

$$[lhs_1, lhs_2, ..., lhs_n] = parallel\_command(rhs_1, rhs_2, ..., rhs_n) ;$$

Right hand side arguments are input parameters and results are returned as the values of the left hand side arguments. These arguments are the standard MATLAB type of complex matrices. When the current command is an invocation of a routine running on the parallel platform, the mex interface is used to pass the rhs arguments to a process on the host computer (Sun workstation). This process converts the matrix data structure used by MATLAB into a message format used by the GENESYS transputer operating system and, acting as a GENESYS process, passes the input arguments to the host node on the parallel platform via the VMEbus or, for remote workstations, over the Ethernet. The host node receives the GENESYS message and routes the rhs arguments to the nodes that this command will use. The compute nodes perform the appropriate function evaluations and return these results to the host node. The host node multi-reels the messages from the compute nodes, performs any arbitration necessary and packages the results into one message for transmission back to the calling process on the host computer. Upon receipt of this message, the host process converts the format of the message into the MATLAB matrix structure and, again acting as a mex process, returns theses matrices as the lhs arguments of the original MATLAB command. In this way the mex-file operates as both a MATLAB function and a GENESYS process having access to the transputer network and is visible to both the parallel platform and the MATLAB environment.

As well as providing the communications interface between MATLAB and the parallel platform, the gateway routines are also used for configuring the transputer network, loading processes onto compute nodes for parallel execution and monitoring the status and performance of the parallel platform. For example,

```
tboot( '1 2 8' ) ;
```

boots an array of 2 by 8 transputers starting at site 1 on the parallel platform. Similarly,

```
[Proc_map Error]= loadtran( '-n2 my_prog' ) ;
```

is used to load *my_prog* onto node 2 of the current network. The return value Proc_map contains information used by another gateway command, exectran, for accessing my_prog. The variable Error is used to indicate the success of the load operation. Other GENESYS OS utilities for loading and killing processes, examining the status of the file system and message passing buffers and freeing compute nodes are also available. Future versions of this gateway are expected to be written using the Inmos C Toolset for greater portabilty.

## 3. Problem Definition

One approach to robot control synthesis is the decomposition of the system into a set of decoupled subsystems, each corresponding to one degree of freedom. This example considers one such subsystem of a manipulator powered by d.c. motors [5]. The model of the subsystem, around the nominal trajectories, is given by:

***Table 1: Model parameters, weighting coefficients and controller parameters for different regimes***

| Regime | $a_{22}$ | $a_{23}$ | $a_{32}$ | $a_{33}$ | $b_3$ | $q_{11}$ | $q_{22}$ | $k_1$ | $k_2$ | J |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | -2.0 | 20.0 | -1000 | -450 | 2000 | 1000 | 1 | -28.03 | -0.754 | 45.94 |
| 2 | -1.33 | 13.3 | -1000 | -450 | 2000 | 2500 | 10 | -39.58 | -2.303 | 177.29 |
| 3 | -1.0 | 10.0 | -1000 | -450 | 2000 | 5000 | 30 | -52.97 | -3.955 | 418.03 |

$$\dot{\underline{x}} = A\underline{x} + \underline{b}u$$

where $\dot{\underline{x}} = \left[q\ \dot{q}\ i_R\right]^T$, q is the angle (displacement) of the joint and $i_R$ is the rotor current and u is the applied voltage. A and $\underline{b}$ take the form

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & a_{22} & a_{23} \\ 0 & a_{32} & a_{33} \end{bmatrix}, \underline{b} = \begin{bmatrix} 0 \\ 0 \\ b_3 \end{bmatrix}$$

The task is to design a robust controller for each subsystem to operate under three important operating regimes corresponding to three different rotor moments of inertia. One approach in such problems is to use the linear quadratic regulator approach to minimise the cost function

$$J = \int_0^\infty (q_{11}x_1^2 + q_{22}x_2^2 + u^2)\, dt$$

for each operating regime. Here, as rotor current sensing is undesirable, we use the suboptimal reduced state feedback formulation where,

$$u = k_1 q + k_2 \dot{q}.$$

Table 1 gives the coefficients of A and b, the corresponding quadratic weighting terms and resulting controller parameters for each regime. The reduced state feedback gains, $k_1$ and $k_2$, are obtained individually for each regime by minimizing $J$, for an average set of initial conditions.

Optimization problems typically involve the minimization of a scalar objective function and can be stated as

$$\min_{\underline{x} \in \Omega} \ f(\underline{x}) \qquad (1)$$

where $x = [x_1, x_2, ..., x_n]^T$ and $\Omega$ defines the set of free variables, x, subject to any constraint functions. Inequality constraints, $g(x) <= 0$, and equality constraints, $h(x) = 0$ may be incorporated into the problem formulation. In casting real world optimization problems into this framework, the designer may have to choose to aggregate many often conflicting design objectives into a single objective function. This causes the distinction between the different design objectives to become obscured and the designer is unable to exercise effective design control in the formulation of the optimization problem.

The use of Multi-objective Optimization (MO) recognises that most practical problems require that a variety of design objectives be satisfied. In contrast to the conventional single-objective optimization problem, equation (1), MO seeks to optimize a number of objectives simultaneously, viz.

$$\min_{\underline{x} \in \Omega} \underline{f}(\underline{x}) \qquad (2)$$

where $\underline{f} = [f_1\ f_2\ \dots\ f_n]^T$ represents a vector whose elements are individual objective functions.

In the solution of (2), it is clear that there will be no one ideal "optimal" solution but rather a set of "non-inferior" or "Pareto-optimal" solutions. A non-inferior solution is one in which an improvement in one of the design objectives will lead to a degradation in one or more of the remaining objectives. The outcome of a MO design process is thus a trade-off surface from which the designer is able to select a suitable compromise solution.

Members of the noninferior solution set are found through solution of an appropriately formulated non-linear programming (NP) problem. Following a survey of alternative strategies, the goal attainment method [4] was selected for use in CACSD problem considered here. Using this method to obtain a noninferior solution for the MO problem, equation (2), the following NP problem is solved

$$\min_{\lambda,\, x \in \Omega} \lambda \qquad (3)$$

such that

$$f_i - w_i \lambda \leq f_i^* \qquad i = 1, 2, \dots, n \qquad (4)$$

where $f_i^*$ are prespecified goals for the design objectives, $f_i$, and $w_i \geq 0$ are designer-selected weights.

Here, minimization of $\lambda$ tends to force the objectives (or design specifications) to meet their goals. It is possible for the designer to set unrealisable goals and still obtain a solution; in this case, the goals will be underattained. The quantity $w_i \lambda$ represents the degree of under- or over-attainment of the corresponding goal, $f_i^*$. The vector w can be visualised as defining the direction of search through the solution space.

Thus, casting the problem as an MO one, we have:-

$$\min_{\underline{k} \in \Omega} \underline{J}$$

where $\underline{J} = [J_1, J_2, J_3]^T$ represents the vector of performance indices for each operating regime and the $\underline{k} = [k_1, k_2]^T$ represents the feedback gain vector to be used in each regime. Recasting the MO formulation in goal attainment form we have:-

$$\min_{k_1, k_2, \lambda} \lambda \quad \text{such that} \quad J_i - w_i \lambda \leq J_i^*,$$

where $J_i^*$ are the optimization goals, set to the optimal performance index values for each regime, and the weighting values $w_i$ are set equal to these values to ensure the same amount of under-attainment in each case. A full description of the solution to this problem can be found in [6].

# 4. Instructions for Use

The demonstration is divided up into a number of sections. Each section starts with a screen of text explaining that part of the demonstration and is identified by a number in the top left hand corner of the screen. Between each section and the individual steps in each section the user is prompted for keyboard input to continue. At any point, the demonstration can be terminated by entering ^C.

Note that this demonstration starts a number of processes external to MATLAB, and links MATLAB to these processes, so care must be taken to ensure that these are terminated gracefully using the appropriate gateway commands (see [2]). If the user terminates the demonstration and leaves MATLAB then the GENESYS command **nt_reset** must be issued to free the site on the parallel platform that this demonstration has reserved.

From the MATLAB command line the parallel processing demonstration is started by entering the script-file name, thus:

```
>> ppdemo
```

## 4.1 Screen 1: Introduction

The opening screen provides an introduction to the demonstration and the problem to be solved.

## 4.2 Screen 2: Multiobjective Optimization

This Screen provides further details of the problem and recasts it in multiobjective goal attainment form.

## 4.3 Screen 3: MATLAB Solution of the Multi-regime Robot Problem

In this Screen, the MATLAB function cmrrp for the evaluation of the cost functions is presented as shown below.

```
function J = cmrrp( K )

A1bar = A_1 + B_1*K*C_1 ;
A2bar = A_2 + B_2*K*C_1 ;
A3bar = A_3 + B_3*K*C_1 ;
temp = C_1'*K'*R_1*K*C_1 ;
Q1bar = Q_1 + temp ;
Q2bar = Q_2 + temp ;
Q3bar = Q_3 + temp ;

J(1) = trace( clyap( A1bar', Q1bar ) ) ;
J(2) = trace( clyap( A2bar', Q2bar ) ) ;
J(3) = trace( clyap( A3bar', Q3bar ) ) ;
```

This function computes the cost function, J, for each regime at the trial solution gain point, K. The routine clyap is a Lyapunov matrix equation solver [7] and trace calculates the sum of the diagonal of the result of clyap. Note that there are three calls made to the same function, each with independent variables and returning a unique result. It is these function calls that will later be evaluated in parallel on the transputer platform.

## 4.4 Screen 4: Optimization

In Screen 4, the optimization routine, attgoal, is used to minimize the cost function presented in Screen 3. First, the model parameters are loaded from disk and made global:

```
load ppdemo
global A_1 A_2 A_3 B_1 B_2 B_3 C_1 Q_1 Q_2 Q_3 R_1
```

The global variables are used because the cost function, cmrrp, can only be passed one set of arguments directly from attgoal, the optimization routine. When these variables are global, they are visible from within the cost function.

The optimization parameters Goal, Weight and the starting point for the search K0 are defined next:

```
Goal = [45.94 177.29 418.03] ;
Weight = Goal ;
K0 = [-39.58 -2.303] ;
```

The optimization itself is invoked as follows (see [8]):

```
K = attgoal( 'cmrrp', K0, Goal, Weight, 1 )
```

and the result obtained

```
K =
      -33.3111        -1.5761
```

This gives the final performance values for the optimized controller as

```
cmrrp( K )
ans =
        48.5166    179.0916    441.4302
```

The final argument of the call to attgoal, 1, is used to report the progress of the optimization.

## 4.5 Screen 5: The Parallel Processing Gateway

Screen 5 is the second part of the demonstration and shows the use of Parallel Processing Gateway and associated routines. Two examples are given: one in which the objectives are evaluated on a single transputer and another mapping each objective onto a separate processor.

The first step in using the gateway is to boot the required network of transputers. The Gateway utility function tboot is used as follows:

```
tboot( 2, 1, 3 )
```

and the following message appears on the user screen:

```
GENESYS booted successfully

1 by 3 network booted from site 2
```

After the parallel platform has been successfully booted, the objective functions can be loaded onto the transputer nodes. The objective functions used in this example use the same code as the C mex-file cmrrp, except that they are compiled with the Gateway compiler. This links the transputer executable code to a shell that emulates the MATLAB mex interface, handling all the communications between MATLAB and the parallel platform. The following command loads the transputer routine glyap onto the nodes booted on the transputer platform. (Further details of the use of loadtran and other Gateway routine and utilities can be found in [2] ).

```
[PM E] = loadtran( 'glyap', 'glyap', 'glyap' )

PM =
        0       543013      18
        1       543013      18
        2       459365      18

E =
        3
        0
```

E is used to determine if the loadtran operation has been successful, the first row reporting the number of loaded routines and the second row reporting the number of errors encountered in loading those routines. PM is a process table that contains in the first column the node identifier, the second column the process id and the last column the index of that process. In loading a routine onto a transputer node, a number of other processes are spawned to provide the MATLAB interface shell and the values returned in PM may be different from those shown. The process numbers shown here may vary according to the current usage and state of the parallel platform. As the process table is needed in the cost function routine it is made global.

```
global PM
```

To examine the state of processors on processes, the utility routine tstate is used.

```
tstate( '0-2' ) ;

NODE     INDEX   PID      KPRI   KSTATE             PROGRAM
n0       [18]    543013   0      BR (543013, 1)     glyap
n1       [18]    543013   0      BR (543013, 1)     glyap
n2       [18]    543013   0      BR (459365, 1)     glyap
```

The information reported by tstate is similar to that contained in the process table. In addition, the priority of the task (KPRI) is reported and its state (KSTATE). Here, KSTATE reports that the processes on all the nodes are blocked awaiting a message with a mask of 1. (Further details concerning tstate and other GENESYS utilities may be found in [9] ).

The cost functions used in this section of the demonstration are the m-files g1mrrp and g3mrrp for evaluating the individual objectives on one and three processors respectively. (These cost functions are reproduced in Appendix A). The next part of the demonstration shows how the routine exectran is used to call C mex-files running on the parallel platform. In the case of the single transputer call the command used is

```
J = exectran( '-T', PM(1,:), A1bar', Q1bar )
```

and for the three transputer case

```
[J1 J2 J3] = exectran('-T',PM(1,:),A1bar',Q1bar,'-T',PM(2,:), ...
A2bar',Q2bar,'-T',PM(3,:),A3bar',Q3bar )
```

In the first example of the demonstration, a single call is made to the routine described in the first column of the process table, in this case node 0 in the network. A1bar and Q1bar are the parameters that would be determined in the m-file g1mrrp, prior to calling the routine. The '-T' in the command line is used to tell exectran that the process is already loaded onto a node and is described in the process table of the next argument. J is used as the destination variable for the result.

The second example follows the same format, except that message are sent to all three nodes, each described in the process table segment passed to exectran with the values of $A_i$bar and $Q_i$bar associated with that call. The return values are stored in the matrices J1, J2 and J3 respectively.

Having ensured that the routines are loaded onto the nodes of the platform, we are now able to call the actual optimization. This uses the same form of the command line as presented in Screen 4, except that the cost function routine name is different reflecting the number of transputers we wish to use. The parameters for the optimization remain the same. The command line for the single transputer evaluation of the cost function is

```
K = attgoal( 'g1mrrp', K0, Goal, Weight, 1 )
```

and the command line for the three transputer case is

```
K = attgoal( 'g3mrrp', K0, Goal, Weight, 1 )
```

The numerical results of the optimization are the same for all of the methods shown in this demonstration. However, it will be noticed that the three processor case ran significantly faster than the single processor example.

The Gateway utility unloadtran is used to remove unwanted processes from processors as follows.

```
unloadtran( PM ) ;
```

If the routine tstate is used then it will be seen that the processes described by the process table PM have been removed from the transputer network. If many sets of routines are being used on the parallel platform at one time, it is possible to have a number of process tables grouping these routines into related families.

## 4.6 Screen 6: Directly coupled concurrent routines

Directly coupled routines differ from the routines called using the parallel processing gateway described in the previous section. Each directly coupled routine is made up of a pair of functions; a MATLAB function that handles the message passing between MATLAB and the transputer platform and a transputer routine for evaluating the appropriate function. The transputer routine may be distributed over a number of processes (see, for example, [10]) or be a single process as used in this demonstration. This Screen of the demonstration is used to show how such routines are used for this example.

The program pairs used in this Screen are tlyap1.c, tlyp1.c and tlyap3.c, tlyp3.c for the one and three transputer examples respectively. The routines tlyap*n*.c are MATLAB C mex-file interface calls and tlyp*n*.c are the transputer routines for the objective function evaluation. Further details about these routines are given in the code listings of Appendix B. As the parallel platform has already been booted in the previous Screen of this demonstration we can load the processes directly onto the parallel platform. This can be done from any shell window on the machine that was used to boot the platform. In this demonstration, we use the MATLAB shell escape feature to call the GENESYS function for loading processes.

```
!loadgo n0 1tlyp
!loadgo n1 3tlyp
```

Again, the state of the processes may be examined using tstate. Now that the routines are running on the platform, they may be called directly from MATLAB using the relevant mex-file interface program. In the optimization call, the cost function names are changed to the m-files containing the calls to these parallel routines. For the single transputer method the optimization call is

```
K = attgoal ( 'tmrrp1', K0, Goal, Weight, 1 )
```

and the three transputer implementation optimization

```
K = attgoal ( 'tmrrp3', K0, Goal, Weight, 1 )
```

Because the gateway has not been used to invoke these function calls, the mex-files communicating with the transputer routines have had to attach themselves directly to the GENESYS operating system. This means that it is not possible to reset or reboot the parallel platform as it will cause MATLAB to crash. The parallel platform must also be reset after the user has exited MATLAB using the GENESYS command tkill. Processes loaded onto the parallel platform can, however, be removed using the unloadtran command by building a processes table based on the information returned by tstate.

### 4.7 Screen 7: Timing Information

The final Screen presents timings for the execution of the various implementations of the example problem. A number of factors will influence these timings, such as the communication loading on the workstation network and the usage of the workstation running MATLAB.

## 5. Concluding Remarks

This report has demonstrated how the parallel processing gateway can be used in CACSD and optimization to distribute numerically intensive cost function routines over a number of nodes on a parallel processing network. In addition, some of the other more salient features of this gateway, such as the process management capabilities have also been reported.

This demonstration has, however, shown that the parallel processing gateway imposes a significant overhead in the communication with tasks on the parallel platform compared with the direct wired routines. This has to be balanced against the ease of use of the gateway and the reduction in the complexity when using more than one set of parallel routines on the same nodes. For example, the user does not have to be concerned with the message passing details of the parallel routines and can run any C mex-file directly on any transputer node. In addition, once the size and number of tasks placed on the parallel platform becomes significant, then the overhead imposed by the gateway routines becomes less important in the total task execution time.

## 6. Acknowledgements

The authors gratefully acknowledge the support of this research by the United Kingdom SERC grant on "Parallel Computing in CACSD" (GR/F94064) and the ESPRIT Parallel Computing Action (project No. PCA4093).

## 7. References

[1] C. Moler, S. Bangert and J. Little *"Pro-MATLAB User's Guide"*, The Mathworks, South Natick, MA 01760, 1990.

[2] A. J. Chipperfield, P. J. Fleming and A. R. Browne, *"Design and use of the MATLAB Parallel Processing Gateway"*, Research Report No. 486, Department of Automatic Control and Systems Engineering, University of Sheffield, UK, 1993.

[3] A.R. Browne, *"A Transparent Parallel Processing Gateway for MATLAB"*, MSc thesis, University of Sheffield, 1992.

[4] F.W. Gembicki, *"Vector Optimization for Control with Performance and Parameter Sensitivity Indices"*, Ph.D. Dissertation, Case Western Reserve University, Cleveland, Ohio, USA,1974.

[5] P.J. Fleming and A.P. Pashkevich, "Computer-aided Control System Design Using a Multiobjective Optimization Approach", *Control '85* Conference, Cambridge, UK, pp. 174-9. 1985.

[6] A. J. Chipperfield, P. J. Fleming, Eds., "MATLAB Toolboxes and Applications for Control", Peter Peregrinus, 1993.

[7] R.H. Bartels and G.W. Stewart, Solution of the Matrix Equation AX + XB = C, *Communications of the ACM*, Volume 15, Number 9, pp820-826, 1972.

[8] T. P. Crummey, R. Farshadnia, P. J. Fleming, A. C. W. Grace and S. D. Hancock, *"An Optimization Toolbox for MATLAB"*, Proceedings IEE Control '91, Edinburgh, UK, Vol. 2, pp. 744-9. 1991.

[9] Genesys C Reference Manual, Transtech Technology, High Wycombe, UK. 1990.

[10] A. J. Chipperfield, P. J. Fleming, D. I. Jones, "Parallel Computing, Multiobjective Optimization and the Workstation CACSD Environment", *Proc. IFAC World Congress '93*, Sydney Australia, 18-23 July 1993.

# Appendix A

# Demonstration Cost Functions

## 1. cmrrp

```
%        Cost function using C mex-file clyap to compute performance
%        index of each regime.

function J = mrrp( K )

A1bar = A_1 + B_1*K*C_1 ;
A2bar = A_2 + B_2*K*C_1 ;
A3bar = A_3 + B_3*K*C_1 ;
temp = C_1'*K'*R_1*K*C_1 ;
Q1bar = Q_1 + temp ;
Q2bar = Q_2 + temp ;
Q3bar = Q_3 + temp ;

J(1) = trace( clyap( A1bar', Q1bar ) ) ;
J(2) = trace( clyap( A2bar', Q2bar ) ) ;
J(3) = trace( clyap( A3bar', Q3bar ) ) ;
```

## 4. ttmrrpl

```
%       Cost function using tlyap1 to call the routine 1tlyp running
%       on node 0 of the parallel platform

function J = tmrrp( K )

A1b = A_1 + B_1*K*C_1 ;
A2b = A_2 + B_2*K*C_1 ;
A3b = A_3 + B_3*K*C_1 ;
temp = C_1'*K'*R_1*K*C_1 ;
Q1b = Q_1 + temp ;
Q2b = Q_2 + temp ;
Q3b = Q_3 + temp ;

J(1) = trace( tlyap1( A1b', Q1b ) ) ;
J(2) = trace( tlyap1( A2b', Q2b ) ) ;
J(3) = trace( tlyap1( A3b', Q3b ) ) ;
```

## 5. tmrrp3

```
%       Cost function using tlyap3 to call 3tlyp on node 0 of the
%       parallel processing platform. 3tlyp passes parameters to
%       nodes 1 and 2 to evaluate all three cost functions concurrently.

function J = tmrrp( K )

A1b = A_1 + B_1*K*C_1 ;
A2b = A_2 + B_2*K*C_1 ;
A3b = A_3 + B_3*K*C_1 ;
temp = C_1'*K'*R_1*K*C_1 ;
Q1b = Q_1 + temp ;
Q2b = Q_2 + temp ;
Q3b = Q_3 + temp ;

[a b c] = tlyap3(A1b',Q1b, A2b',Q2b, A3b',Q3b) ;
J = [trace(a), trace(b), trace(c) ] ;
```

# Appendix B

## Example MATLAB-Transputer Interface

### 1. tlyap1.c

```
/*
 *      MATLAB mex-file to pass 2 square input matrices of order n to node 0 on
 *      the parallel platform and receive back 1 square order n matrix. Meant for
 *      use with the Lyapunov matrix equation solver tlyp1.c
 */
#include    <genesys/typical.h>
#include    <genesys/net.h>
#include    <math.h>
#include    <cmex.h>

#define     MESG     100

user_fcn( nlhs, plhs, nrhs, prhs )
int         nlhs, nrhs ;
Matrix      *plhs[], *prhs[] ;
{
            struct      nmsg      header ;
            int         i, j, n, m ;
            double      *a, *b, *message ;

            /* check input args */
            if( nrhs != 2 )
                        mex_error( "tlyap1 requires two input arguments" ) ;

            if( (prhs[0]->m != prhs[0]->n) || (prhs[0]->m != prhs[0]->n) )
                        mex_error( "clyap requires square input matrices\n" ) ;

            /* i/p args appear OK, attach to Genesys */
            if (kinit(0)) _exit(0) ;

            /* copy A' and B into message structure */
            n = prhs[0]->n ; m = n * n ;
            a = prhs[0]->pr ; b = prhs[1]->pr ;
            message = (double *)malloc( 2 * m * sizeof( double ) );
            for( i = 0 ; i < n ; ++i )
                        for( j = 0 ; j < n ; ++j ) {
                                    message[(i*n)+j] = a[(j*n)+i] ;
                                    message[(i*n)+j+m] = b[(i*n)+j] ;
                        }

            /* send to T0 */
            header.nh_event = MESG ;
            header.nh_type = 0 ;
            header.nh_flags = DDBLMSG ;
            header.nh_length = 2 * m * sizeof( double ) ;
            header.nh_data[0] = n ;
            header.nh_msg = (char *) message ;
            header.nh_node = 0 ;
            nsend ( &header )

            /* receive reply and pass back X to matlab */
            plhs[0] = create_matrix( n, n, REAL );

            header.nh_event = MESG + 1;
            header.nh_type = 0 ;
            header.nh_flags = DDBLMSG ;
            header.nh_length = 4096 ;
            header.nh_msg = (char *) plhs[0]->pr ;
            nrecv ( &header )
}
```

## 2. tlyp1.c

```
/*
 *          Lyapunov matrix equation solver for use with tlyap1.c. Receives two order
 *          n input matrices, A and B, computes solution of A'X + XA = -B and returns
 *          the result to a receiving process on the host computer
 */
#include    <genesys/typical.h>
#include    <genesys/net.h>
#include    <math.h>

#define     MESG      100

main()
{
            int           i, j, n, m, ierr ;
            double *message, *a, *b, *A, *B, *wr, *wi, *Z ;
            struct      nmsg        header ;

            message = (double *) malloc( 512 * sizeof( double ) );

            while( 1 ) {              /* Loop Continually */

                      /* Receive message from host (or elsewhere) */
                      header.nh_event = MESG ;
                      header.nh_type = 0 ;
                      header.nh_flags = DDBLMSG ;
                      header.nh_length = 4096 ;
                      header.nh_msg = (char *) message ;
                      nrecv( &header )
```

```
                /* get order of input arguments */
                n = (int) header.nh_data[0] ;
                m = n * n ;

                /* Assign temporary storage space */
                wr = (double *) malloc( n * sizeof( double ) ) ;
                wi = (double *) malloc( n * sizeof( double ) ) ;
                A = (double *) malloc( n * n * sizeof( double ) ) ;
                B = (double *) malloc( n * n * sizeof( double ) ) ;
                Z = (double *) malloc( n * n * sizeof( double ) ) ;

                /* Copy A and B into local storage space */
                for( i = 0 ; i < m ; ++i ) { /* Get A and B */
                        A[i] = message[i] ;
                        B[i] = message[i+m] ;
                }

                /* Compute Schur form of A */
                orthes( n, 1, n, A, wr ) ;
                ortran( n, n, 1, n, A, wr, Z ) ;
                hqrort( n, n, 1, n, A, wr, wi, Z, &ierr ) ;

                /* Solve for X , answer stored in B */
                mqfwo( n, n, n, B, Z, wr ) ;
                symslv( n, n, n, A, B ) ;
                trnata( n, Z ) ;
                mqfwo( n, n, n, B, Z, wr ) ;
                trnata( n, Z ) ;

                /* Return result to host */
                header.nh_event = MESG + 1;
                header.nh_type = 0 ;
                header.nh_flags = DDBLMSG ;
                header.nh_length = m * sizeof( double ) ;
                header.nh_msg = (char *) B ;
                header.nh_node = HOST ;
                nsend( &header )

                /* free temporary storage */
                free( wr ) ; free( wi ) ; free( A ) ; free( B ) ; free( Z ) ;
        }
}
```