Pergamon

PII: S0005-1098(97)00053-8

Automatica, Vol. 33, No. 8, pp. 1579–1586, 1997
© 1997 Elsevier Science Ltd. All rights reserved
Printed in Great Britain
0005-1098/97 $17.00 + 0.00

Brief Paper

Basic Hardware Module for a Nonlinear Programming Algorithm and Applications*

SHIN-YEU LIN†

Key Words—Nonlinear programming; optimal control; nonlinear systems; nonlinear control; implementations; integrated circuits.

Abstract—We present a VLSI-array-processor architecture for the implementation of a nonlinear programming algorithm that solves discrete-time optimal control problems for nonlinear systems with control constraints. We incorporate this hardware module with a two-phase parallel computing method and develop a VLSI-array-processor architecture to implement a receding horizon controller for constrained nonlinear systems. On the basis of current VLSI technologies, the estimated computing time to obtain the receding-horizon feedback-control solution meets the real-time processing-system needs. © 1997 Elsevier Science Ltd.

1. *Introduction*

Discrete-time optimal control problems for nonlinear systems with control constraints are popular control problems. Numerous numerical techniques (Dyer and McReynolds, 1970) have been developed for solving this type of problem; however, computational efficiency is still a major issue that is frequently encountered in the solution methods. In this paper, we shall propose an algorithm that combines a projected Jacobi method and a Lagrange-dual Jacobi method to solve discrete-time optimization problems for nonlinear systems, which include discrete-time optimal control problems with control constraints. Our algorithm will achieve a complete decomposition effect, and the data and command flow within the algorithm are very simple and regular; these two factors suggest that our algorithm can be implemented by VLSI array processors to meet real-time processing-system needs. Thus the first contribution of this paper is the presentation of a VLSI-array-processor architecture for the implementation of our nonlinear programming algorithm. Implementing a computational algorithm by VLSI array processors to improve the computational efficiency has been a trend, especially in the area of signal processing; for example, Frantzeskakis and Liu (1994) deal with a least-squares problem with linear equality constraints. However, the nonlinear programming problem considered in this paper is more complicated because of the presence of nonlinearity and inequality constraints.

As well as applications to solving discrete-time optimal control problems for nonlinear systems, our hardward implementable algorithm has important applications to receding-horizon control. In recent years, there has been a growing interest for the design of receding-horizon feedback controllers (Mayne and Michalska, 1990a; Clarke and Scattolini, 1991; Mayne and Polak, 1993; Richalet, 1993; DeNicolao *et al.*, 1996). For such controllers, stability is guaranteed for the zero-terminal-state strategy (Mosca and Zhang 1992; DeNicolao and Scattolini, 1994). The most distinguished characteristic of this controller compared with other control methodologies is its global stability for general nonlinear systems, as shown by Mayne and Michalska (1990a, b, 1991); however, this is at the expense of high computational complexity to obtain a control solution. Although model reduction is an attractive approach to reduce the computational complexity (Richalet, 1993), this approach may not apply to a general, especially a highly nonlinear, system. Thus, to cope with this computational difficulty, we have proposed a two-phase parallel computing method (Lin, 1993, 1994) to obtain the solution for receding-horizon feedback control. The phase 1 method uses a two-level (master- and slave-level) approach to solve a feasibility problem to obtain an admissible control and horizon pair. The control solution obtained in phase 1 is improved by phase 2, and the final solution is taken as the receding-horizon feedback control solution for the current sampling interval. The problems formulated in this two-phase method, except for the master-level problem in phase 1, are discrete-time optimization problems for nonlinear systems. Thus we can use our hardware-implementable algorithm as a basic algorithm module in the two-phase method, and this results in a simpler algorithm than that in Lin (1994) for solving a receding-horizon feedback control solution. Since the master problem in phase 1 can be solved by simple calculations, this suggests that the two-phase method can be implemented by VLSI array processes. Thus presenting a VLSI-array-processor architecture for a receding-horizon feedback controller is the second contribution of this paper.

Since receding-horizon control is one of the most promising globally stabilizing control methodologies for highly nonlinear systems, the work described in this paper also represents an effort to realize a real-time controller for nonlinear systems.

2. *Basic hardware module for a nonlinear programming algorithm*

2.1. *Statement of the the discrete-time optimization problem of a nonlinear system.* We consider discrete-time optimization problems for nonlinear systems of the form

$$\begin{aligned} &\min_{y} \sum_{i=0}^{N} y_i^T M_i y_i, \\ &y_{i+1} - Ky_i + p(y_i) = 0, \quad i = 0, \ldots, N-1, \\ &\psi(y_N) = 0, \quad \underline{y}_i \le y_i \le \bar{y}_i, \quad i = 0, \ldots, N, \end{aligned} \tag{1}$$

where the $n \times n$ matrix M_i is positive-semidefinite, $y = (y_0, \ldots, y_N)$, $y_i \in \mathbb{R}^n$, $i = 0, \ldots, N$, are variables to be solved, K is an $n \times n$ constant matrix, $p(y_i)$ is a n-dimensional vector function of y_i, $\psi(y_N) = 0$ represents the terminal constraints and ψ is a q-dimensional vector function, $\bar{y}_i$ and $\underline{y}_i$ represent the upper and lower bound respectively of the variables y_i. The discrete-time optimal control problem

* Received 15 June 1995; revised 3 June 1996; received in final form 31 January 1997. This paper was presented at the 3rd IFAC Symposium on Nonlinear Control System Design (NOLCOS), which was held in Tahoe City, CA during 25–28 June 1995. The Published Proceedings of this IFAC meeting may be ordered from: Elsevier Science Limited, The Boulevard, Langford Lane, Kidlington, Oxford OX5 1GB, U.K. This paper was recommended for publication in revised form by Associate Editor Matthew James under the direction of Editor Tamer Başar. Corresponding author Professor Shin-Yeu Lin. Tel. +886 35 731839; Fax +886 35 715998; E-mail sylin@cc.nctu.edu.tw.

† Department of Control Engineering, National Chiao Tung University, Hsinchu, Taiwan.

for a nonlinear system with quadratic objective function and simple control constraints is a case of (1) under the conditions that

$$y_i = (x_i, y_i), \quad K = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}, \quad p(y_i) = \begin{bmatrix} f(x_i, u_i) \\ 0 \end{bmatrix},$$
$$\bar{y}_i = \begin{bmatrix} \infty \\ \bar{u}_i \end{bmatrix}, \quad \underline{y}_i = \begin{bmatrix} -\infty \\ \underline{u}_i \end{bmatrix},$$

where I denotes the identity matrix and 0 denote the zero submatrix or zero vector function with appropriate dimensions; $x_{i+1} - x_i - f(x_i, u_i) = 0$ is the system dynamic equation. Note that the formulation in (1) can include the case of general control constraints; for the sake of simplicity, it is not discussed here, but the explanation can be found in Section 3.

2.2. *The algorithm.* To solve (1), we propose an algorithm that combines the projected Jacobi method with the Lagrange-dual Jacobi method. The projected Jacobi method uses the following iterations:

$$y_i(l+1) = y_i(l) + \gamma_1 \, dy_i^*(l), \quad i = 0, \ldots, N, \tag{2}$$

where γ_1 is a step size and l is the iteration index. Here $dy_i^*(l)$ is the solution of the following quadratic subproblem:

$$\begin{aligned} &\min_{y} \sum_{i=0}^{N} dy_i^{\mathrm{T}} \hat{M}_i dy_i + [M_i y_i(l)]^{\mathrm{T}} dy_i, \\ &E_i(l) + dy_{i+1} - K dy_i + p_{y_i}(l) dy_i = 0, \quad i = 0, \ldots, N-1 \\ &\psi(y_N(l)) + \psi_{y_N} d_{y_N}(l) dy_N = 0 \\ &\underline{y}_i \le y_i(l) + dy_i \le \bar{y}_i, \quad i = 0, \ldots, N, \end{aligned} \tag{3}$$

where $E_i(l) = y_{i+1}(l) - K y_i(l) + p(y_i(l))$, $p_{y_i}(l)$ is the derivative of the vector function p with respect to y_i at $y_i(l)$, $\psi_{y_N}(l)$ is the derivative of ψ with respect to y_N at $y_N(l)$, dy_i, $i = 0, \ldots, N$, are variables to be solved, and the diagonal matrix $\hat{M}_i$ is formed by setting the kth diagonal element m_k as

$$m_k = \begin{cases} M_{kk} & \text{if } M_{kk} > \delta, \\ \delta & \text{otherwise,} \end{cases}$$

where M_{kk} is the kth diagonal element of M_i and δ is a small positive real number.

It can be verified that if $y_i(l)$, $i = 0, \ldots, N$, is feasible then $dy_i^*(l)$, $i = 0, \ldots, N$, is a descent direction of (1); $y_i(l+1)$, $i = 0, \ldots, N-1$, is also feasible if $0 < \gamma_1 \le 1$. Thus the projected Jacobi method is a descent iterative method, which converges and solves (1) if γ_1 is small.

Since (3) is a quadratic programming problem with strictly convex objective function, by the strong duality theorem (Bazaraa and Shetty, 1979), we can solve the corresponding dual problem instead of solving (3) directly.

The dual problem of (3) is

$$\max_{\lambda} \phi(\lambda), \tag{4}$$

where the dual function

$$\begin{aligned} \phi(\lambda) = \min_{\underline{y} \le y(l) + dy \le \bar{y}} \sum_{i=0}^{N-1} &\{dy_i^{\mathrm{T}} \hat{M}_i dy_i + [M_i y_i(l)]^{\mathrm{T}} dy_i \\ &+ \lambda_i^{\mathrm{T}} [E_i(l) + dy_{i+1} - K dy_i + p_{y_i}(l) dy_i]\} \\ &+ dy_N^{\mathrm{T}} \hat{M}_N dy_N + [M_N y_N(l)]^{\mathrm{T}} dy_N \\ &+ \lambda_N^{\mathrm{T}} [\psi(y_N(l)) + \psi_{y_N}(l) dy_N] \end{aligned} \tag{5}$$

is a function of the Lagrange multiplier $\lambda_i \in \mathbb{R}^n$, $i = 0, \ldots, N$. We can use the following Lagrange-dual Jacobi method to solve (4):

$$\lambda_i(t+1) = \lambda_i(t) + \gamma_2 \, \Delta\lambda_i(t), \quad i = 0, \ldots, N, \tag{6}$$

where γ_2 is a step size, and $\Delta\lambda(t) = (\Delta\lambda_0(t), \ldots, \Delta\lambda_N(t))$ is obtained by solving the linear equations

$$\operatorname{diag}[\nabla^2_{\lambda\lambda} \phi^u(\lambda(t))] \, \Delta\lambda(t) + \nabla_\lambda \phi(\lambda(t)) = 0. \tag{7}$$

Here $\phi^u(\lambda)$, the unconstrained dual function of ϕ, is defined by relaxing the inequality constraints on primal variables of ϕ as follows:

$$\begin{aligned} \phi^u(\lambda) = \min \sum_{i=0}^{N-1} &[\{dy_i^{\mathrm{T}} \hat{M}_i dy_i + [M_i y_i(l)]^{\mathrm{T}} dy_i \\ &+ \lambda_i^{\mathrm{T}} [E_i(l) + dy_{i+1} - K dy_i + p_{y_i}(l) dy_i]\} \\ &+ dy_n^{\mathrm{T}} \hat{M}_N dy_N + [M_N y_N(l)]^{\mathrm{T}} dy_N \\ &+ \lambda_N^{\mathrm{T}} [\psi(y_N(l)) + \psi_{y_N}(l) dy_N], \end{aligned} \tag{8}$$

where $\nabla^2_{\lambda\lambda} \phi^u(\lambda)$ is the Hessian matrix of $\phi^u(\lambda)$, $\operatorname{diag}[\nabla^2_{\lambda\lambda} \phi^u(\lambda)]$ is a diagonal matrix whose diagonal elements are taken from the diagonal elements of $\nabla^2_{\lambda\lambda} \phi^u(\lambda)$, and $\nabla_\lambda \phi(\lambda)$ is the gradient of ϕ with respect to λ. Based on Luenberger (1984), $\nabla_\lambda \phi(\lambda)$ and $\nabla^2_{\lambda\lambda} \phi^u(\lambda)$ can be computed by

$$\nabla_{\lambda_i} \phi(\lambda(t)) = \begin{cases} E_i(l) + \hat{d}y_{i+1}(\lambda(t)) - K d\hat{y}_i + p_{y_i}(l) \hat{d}y_i(\lambda(t)), \\ \qquad i = 0, \ldots, N-1, \\ \psi(y(l)) + \psi_{y_N}(l) \hat{d}y_N, \; i = N, \end{cases} \tag{9}$$

$$\begin{aligned} &\operatorname{diag}[\nabla^2_{\lambda_i \lambda_i} \phi^u(\lambda(t))] \\ &= \begin{cases} -\operatorname{diag}\{[(-K + p_{y_i}(l)) \hat{M}_i^{-1} [-K + p_{y_i}(l)]^{\mathrm{T}} + \hat{M}_{i+1}^{-1}\}, \\ \qquad i = 0, \ldots, N-1 \\ -\operatorname{diag}[\psi_{y_N}(l) \hat{M}_N^{-1} \psi_{y_N}(l)^{\mathrm{T}}], \; i = N, \end{cases} \end{aligned} \tag{10}$$

where $\hat{d}y_i(\lambda(t)) \in \mathbb{R}^n$, $i = 0, \ldots, N$, in (9) is the solution of the constrained minimization problem on the right-hand side of (5) with $\lambda = \lambda(t)$. Since $\operatorname{diag}[\nabla^2_{\lambda\lambda} \phi^u(\lambda(t))]$ is a diagonal matrix, we can compute $\Delta\lambda(t)$ from (7) analytically by

$$\Delta\lambda_i(t) = -\{\operatorname{diag}[\nabla^2_{\lambda_i \lambda_i} \phi^u(\lambda(t))]\}^{-1} \nabla_{\lambda_i} \phi(\lambda(t)), \quad i = 0, \ldots, N. \tag{11}$$

It can easily be verified from (10) that $\operatorname{diag}[\nabla^2_{\lambda\lambda} \phi_u(\lambda(t))]^{-1}$ is negative-definite. Thus $\Delta\lambda_i(t)$, $i = 0, \ldots, N$, obtained from (11) is an ascent direction of (4). Then the Lagrange-dual Jacobi method (6) will converge and solve (4) provided that γ_2 is small.

However, to calculate $\nabla_{\lambda_i} \phi(\lambda(t))$, we need the value of $\hat{d}y_i(\lambda(t))$, $i = 0, \ldots, N$, which can be found by the following two steps. First, we solve for the solution $\tilde{d}y_i$, $i = 0, \ldots, N$, of the unconstrained minimization problem on the right-hand side of (8), which can be obtained analytically by

$$\begin{aligned} &\tilde{d}y_i(\lambda(t)) \\ &= \begin{cases} \hat{M}_i^{-1} \{-M_i y_i(l) - \lambda_{i-1} + [K - p_{y_i}(l)]^{\mathrm{T}} \lambda_i\}, & i = 0, \ldots, N-1 \\ \hat{M}_N^{-1} [-M_N y_N(l) - \lambda_{N-1} - \psi_{y_N}(l)^{\mathrm{T}} \lambda_N], & i = N. \end{cases} \end{aligned} \tag{12}$$

Then, we project $\tilde{d}y_i$, $i = 0, \ldots, N$, to the constraint set $\underline{y} \le y(l) + dy \le \bar{y}$, and the resulting projection $\hat{d}y_i$, $i = 0, \ldots, N$, can be obtained analytically by

$$\hat{d}y_i(\lambda(t)) = \begin{cases} \tilde{d}y_i(\lambda(t)) & \text{if } \underline{y}_i \le y_i(l) + \tilde{d}y_i(\lambda(t)) \le \bar{y}_i, \\ \bar{y}_i - y_i(l) & \text{if } y_i(l) + \tilde{d}y_i(\lambda(t)) > \bar{y}_i, \\ \underline{y}_i - y_i(l) & \text{if } y_i(l) + \tilde{d}y_i(\lambda(t)) < \underline{y}_i, \end{cases} \tag{13}$$

$i = 0, \ldots, N$. It can easily be verified that $\hat{d}y_i$, $i = 0, \ldots, N$ obtained from (12) and (13) are indeed the solutions of the constrained minimization problem on the right-hand side of (5).

2.2.1. *The complete decomposition property.* In our algorithm, the projected Jacobi method performs only an update procedure in (2); all the major calculations lie in the iterations of the Lagrange-dual Jacobi method (6) for solving (4). Starting from a given $\lambda(t)$, the Lagrange-dual Jacobi method solves $\hat{d}y_i(\lambda(t))$, $i = 0, \ldots, N$, from (12) and (13), then computes $\nabla_{\lambda_i} \phi(t))$ and $\nabla^2_{\lambda_i \lambda_i} \phi^u(\lambda(t))$, $i = 0, \ldots, N$, by (9) and (10) respectively. It then calculates $\Delta\lambda_i(t)$, $i = 0, \ldots, N$, from (11), updates $\lambda_i(t)$, $i = 0, \ldots, N$, by (6) and proceed with the next iteration. This iterative process will continue until convergence occurs. From (6) and (9)–(13), we see that a complete decomposition effect has been achieved by our algorithm. This property results from (3) being separable.

2.2.2. *The algorithm steps.*

Step 1. Initially guess the values of $y(0)$ and set $l = 0$.

Step 2. Compute $E_i(l)$, $p_{y_i}(l)$, $i = 0, \ldots, N-1$, $\psi(y_N(l))$, and $\psi_{y_N}(l)$.

Step 3. Initially guess the value of $\lambda(0)$, or set it as the previous value, and set $t = 0$.

Step 4. Solve $\hat{d}y_i(\lambda(t))$, $i = 0, \ldots, N$, by (12) and (13).

Step 5. Compute $\nabla_{\lambda_i}\phi(\lambda(t))$ by (9), and diag $[\nabla^2_{\lambda_i\lambda_i}\phi''(\lambda(t))]$ by (10), $i = 0, \ldots, N$.

Step 6. Compute $\lambda_i(t+1) = \lambda_i(t) - \gamma_2[\text{diag}\,(\nabla^2_{\lambda_i\lambda_i}\phi'')]^{-1}\nabla_{\lambda_i}\phi(\lambda(t))$, $i = 0, \ldots, N$. Check whether $\|\Delta\lambda(t)\|_\infty < \varepsilon$; if yes, go to Step 7; otherwise, set $t = t+1$ and return to Step 4.

Step 7. Set $dy_i^*(l) = \hat{d}y_i(\lambda(t))$ and update $y_i(l+1) = y_i(l) + \gamma_1 dy_i^*(l)$, $i = 0, \ldots, N$. Check whether $\|dy^*(l)\|_\infty < \varepsilon$; if yes, stop; otherwise, set $l = l+1$ and return to Step 2.

2.3. *The VLSI-array-processor architecture for implementing the algorithm.* Since both $\hat{M}_i$ and diag $[\nabla^2_{\lambda_i\lambda_i}\phi''(\lambda)]$ are diagonal matrices, $\hat{M}_i^{-1}$ and $[\text{diag}\,(\nabla^2_{\lambda_i\lambda_i}\phi''(\lambda)]^{-1}$ can be computed analytically. Thus all the computations required in our algorithm steps are simple arithmetic operations, and are independent with each other on different time intervals owing to the complete decomposition property. This motivates us to implement the proposed algorithm using VLSI array processors by assigning a processing element (PE) to the computation required in a time interval of an algorithm step.

2.3.1. *Modification of the convergence criteria.* Since our algorithm converges, the $\Delta\lambda(t)$ in Step 6 and the $dy^*(l)$ in Step 7 will approach zero as the number of iterations t and l increase. Thus, instead of using a tolerance of accuracy, ε, for convergence criteria in Steps 6 and 7, we may assign an arbitrary number of iterations $t_{\max}$ for the Lagrange-dual Jacobi method and $l_{\max}$ for the projected Jacobi method, and modify the convergence criteria in Steps 6 and 7 as follows.

Step 6(m). ... If $t \geq t_{\max}$, go to Step 7; ....

Step 7(m). ... If $l \geq l_{\max}$, stop; ....

2.3.2. *The mapping of the algorithm steps to the VLSI-array-processor architecture.* Suppose we assign one PE for performing the computation of an algorithm step in a time interval; all the PEs should be linked so that the data and command flows in between PEs can make the PE arrays to perform the algorithm just as in a sequential computer. Thus the construction of the array-processor architecture should be based on the data and command flow in the algorithm steps. Examples of data flows are as follows: the data $\hat{d}y_i(\lambda(t))$ and $\hat{d}y_{i+1}(\lambda(t))$ computed in Step 4 are needed in the computation of $\nabla_{\lambda_i}\phi(\lambda(t))$ in Step 5; the data $\nabla_{\lambda_i}\phi(\lambda(t))$ and $\nabla^2_{\lambda_i\lambda_i}\phi''(\lambda(t))$ computed in Step 5 are needed in the computation of $\lambda_i(t+1)$ in Step 6(m). The command flow is more complicated; for example, in Step 6(m), if the Lagrange-dual Jacobi method converges, the data $\hat{d}y_i(\lambda(t))$ computed in Step 2 should be transferred to Step 7(m). This is a procedure of data flow followed by a command flow. In our algorithm, there are two types of commands: one is the initial-value request in Steps 1 and 3; another is the notification of convergence in Steps 6(m) and 7(m).

Figure 1 shows the VLSI-array-processor architecture for implementing our algorithm; for the sake of simplicity, but without loss of generality, we let $N = 3$ in Fig. 1. Each square block in Fig. 1 denotes a PE. The PEs lying in the same array will perform the same algorithm step. The directed solid links denote the data-transfer path. The directed dash-dotted links also denote the data-transfer path; however, they differ from the solid links in that receiving PEs will not use the transferred data for computation immediately. The directed dotted links in Fig. 1 denote the flow of commands of initial-value request or convergence notification. The arrows of these three types of links describe the data and command flows in the architecture.

In the following, we shall explain the mapping of the algorithm steps to the architecture with the aid of Table 1. In the first column of the table, we indicate the type and the corresponding time interval of a PE by superscript and subscript respectively. The second column lists the corresponding algorithm step of each PE, which means that the computations in a time interval or a logical decision for convergence check carried out in an algorithm step will be performed in the corresponding PE. For example, PE_2^3 will compute $\nabla_{\lambda_2}\phi(\lambda(t))$ and diag $[\Delta^2_{\lambda_2\lambda_2}\phi''(\lambda(t))]$. Although Steps 1 and 3 concerning the initial-value guesses do not require any computation, they will be taken care of by PE_i^1 and PE_i^2 respectively. We shall explain how the initial values are provided when we introduce columns 5 and 6 of Table 1. Thus each algorithm step has a corresponding PE. The third and fourth columns show the output data and the corresponding destination of each PE, where the output data of a PE are its computed data. These two columns explain

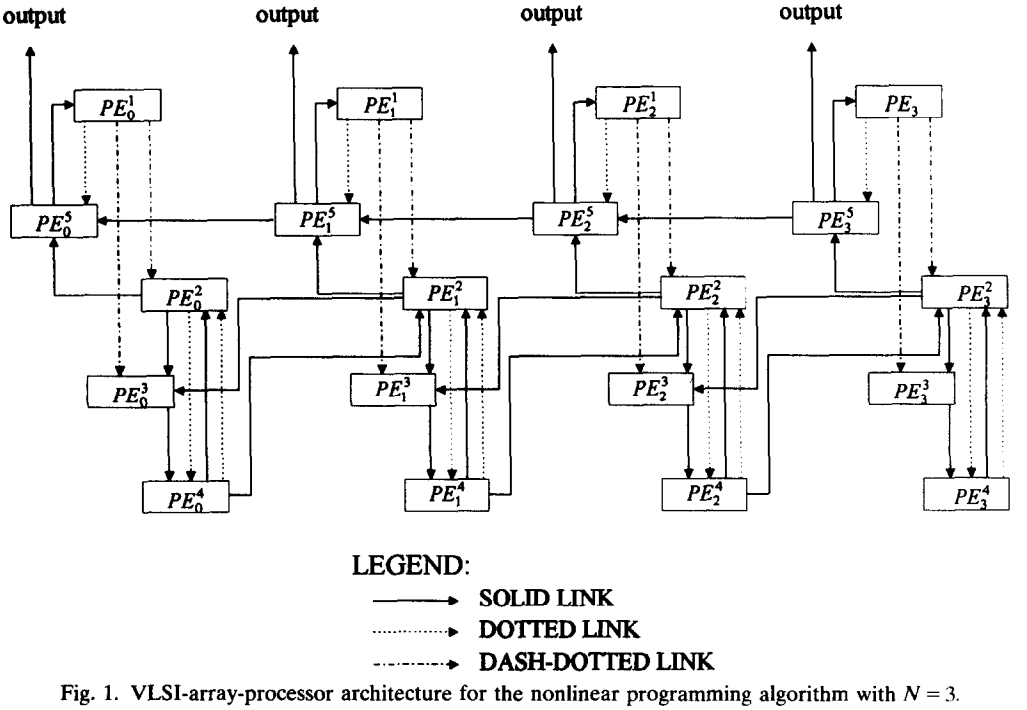


Fig. 1. VLSI-array-processor architecture for the nonlinear programming algorithm with $N = 3$.

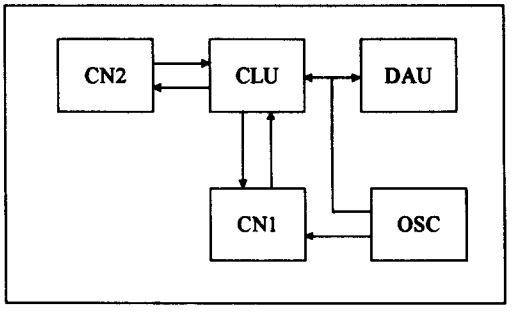
Table I. The characteristics of PE

PE^{type}_{time}	Algorithm Step	Output data	Destination of output data	Output command	Destination of output command	Time complexity
PE_i^1	1,2	$E_i(l)$, $p_{y_i}(l)$ if $i \neq N$, $\psi(y_N(l))$, $\psi_{y_N}(l)$ if $i = N$	PE_i^2 for $p_{y_i}(l)$, PE_i^3 for $E_i(l)$	Request of initial guess if $l = 0$	PE_i^5	Unknown
PE_i^2	3,4	$\hat{d}y_i(\lambda(t))$	PE_i^5 if $t = t_{max}$ in PE_i^4, else, PE_i^3 and PE_{i-1}^3	Request of initial guess if $l = 0$, $t = 0$	PE_i^4	$2\otimes$ and $\log_2(2n+2)\oplus$
PE_i^3	5	$\nabla_{\lambda_j}\phi(\lambda(t))$, $\text{diag}[\nabla^2_{\lambda_i\lambda_i}\phi^a(\lambda(t))]$	PE_i^4	—	—	$2\otimes$ and $\log_2 n^2\oplus$
PE_i^4	6(m)	$\lambda_i(t+1)$	PE_i^2 and PE_{i+1}^2	Convergence if $t = t_{max}$	PE_i^2	$2\otimes$ and $\log_2(n+1)\oplus$
PE_i^5	7(m)	$y_i(l+1)$	Halt if $l = l_{max}$ in PE_i^5; else, PE_i^1	—	—	$1\oplus$

the mapping of the data flow in the algorithm to the architecture, as described by the following examples. The data $E_i(l)$ (or $\psi(y_N(l)$ if $i = N$) computed in PE_i^1 is sent to PE_i^3, and the data $p_{y_i}(l)$ (or $\psi_{y_N}(l)$ if $i = N$) is sent to PE_i^2. Since these data will not be used for computation immediately, the data flows are indicated by dash-dotted links directed from PE_i^1 to PE_i^3 and PE_i^1 to PE_i^2 in Fig. 1. This corresponds to the data flow from Step 2 to Step 5. The output data $\hat{d}y_i$ computed in PE_i^2 is sent to PE_i^3 and PE_{i-1}^3, which is indicated by solid links directed from PE_i^2 to PE_i^3 and PE_{i-1}^3 in Fig. 1. This corresponds to the data flow from Step 4 to Step 5. However, the data $\hat{d}y_i$ will be sent to PE_i^5 if $t = t_{max}$; this is a situation of data flow followed by a command flow. The mapping of the data flows from Step 5 to Step 6, from Step 6 to Step 4 and from Step 7 to Step 2 can also be observed from the third and fourth columns of Table 1 and the directed solid links shown in Fig. 1. Columns 5 and 6 show the output commands and corresponding destinations of each PE. There are two types of commands: one is the request for initial guesses and another is the notification of convergence. As we have described earlier, Steps 1 and 3 concerning the initial guesses will be taken care of by PE_i^1 and PE_i^2 respectively. These steps are performed as follows. When $l = 0$, Step 1 needs an initial guess for $y(l)$, and when $l = 0$ and $t = 0$, Step 3 needs an initial guess for $\lambda(t)$. Therefore, when $l = 0$, PE_i^1 will output the command of initial-value request to PE_i^5, which will respond by sending a default value of $y_i(l)$ to PE_i^1. This command flow is described in columns 5 and 6 of the second row, and is indicated by the dotted link directed from PE_i^1 to PE_i^5 in Fig. 1. A similar situation occurs for PE_i^2 to request the initial value of $\lambda(t)$ from PE_i^4 when $l = 0$ and $t = 0$, as indicated in columns 5 and 6 of the third row in Table 1 and the dotted links directed from PE_i^2 to PE_i^4 in Fig. 1. For the convergence command, we see that if the iteration index $t = t_{max}$ is detected in PE_i^4, it will output a command of convergence to PE_i^2, as described in columns 5 and 6 of the fifth row. Then PE_i^2 will respond by sending the data $\hat{d}y_i(\lambda(t))$ to PE_i^5, as described in columns 3 and 4 of the third row in Table 1, which has not yet been explained. As shown in Fig. 1, this command flow is indicated by the dotted links directed from PE_i^4 to PE_i^2, and the data flow followed by receiving the command is indicated by the solid links directed from PE_i^2 to PE_i^5. These command and data flows represent the mapping of the command and data flows from Step 6 to Step 7. However, when PE_i^5 detects $l = l_{max}$, it will halt the execution and output the solution, as described in columns 3 and 4 in the sixth row, which has not yet been explained.

2.4. *Timing of the computations of the VLSI array processors.* When using VLSI array processors to perform the algorithm, synchronization is necessary. In general, a global clock will cause severe time delay. Thus, to circumvent the drawback of global clock and maintain the synchronization, the data-driven-computation PE (Kung, 1988) associated with an asynchronous handshaking communication link (Kung, 1988) for data and command flows can be the solution. Therefore the computation in each PE will be activated after the completion of all the data transfers from the solid links; this will ensure that the computations in PEs lying in the same array are carried out asynchronously and simultaneously. Nevertheless, a self-timed clock is needed in each PE to control the synchronization of the operations in each individual PE.

2.5. *Realization of PEs and time complexity.* Basically, each PE consists of a self-timed clock, a control logic unit, two counters and a dedicated arithmetic unit. The typical structure of a PE is shown in Fig. 2. The self-timed clock is used to control the synchronization of the operations within the PE. The dedicated arithmetic unit may consist of multipliers, adders and various types of registers. Counter #1 in Fig. 2 is used to count clock pulses in order to indicate the completion of the arithmetic operations. Counter #2 is available only in PE_i^4 and PE_i^5 for each i, and detects whether $t = t_{max}$ or $l = l_{max}$ in the Lagrange-dual Jacobi method and the projected Jacobi method respectively. Note that once Counter #2 has reached the value of t_{max} in PE_i^4 or l_{max} in PE_i^5, it will be reset for next count of clock pulse. The functions of the control logic unit include the control of the sequence of arithmetic operations and the timing of the right communication link for sending out the data and the



LEGEND:
DAU - dedicated arithmetic unit
CLU - control logic unit
CN1 - counter #1
CN2 - counter #2
OSC - oscillator

Fig. 2. Typical structure of the processing elements.

reactions to the input command. For example, as shown in columns 3 and 4 of the third row of Table 1, and control logic circuitry in PE_i^2 should determine which of the following solid links should be activated based on the value of the iteration index appearing in Counter #2: the solid link directed to PE_i^5 or the solid links directed to PE_i^3 and PE_{i-1}^3.

According to column 2 of Table 1 and the details of the algorithm steps, the structure of the dedicated arithmetic units of each PE can easily be realized by logical curcuitry and arithmetic units. For example, the formulas (12) and (13) for the computation of one component $\hat{d}y_i$, say $\hat{d}y_i^j$, in PE_i^2 can be realized as in Fig. 3, in which part of the multiplexer is used to perform the projection (13). From Fig. 3, we can derive the time complexity of the computations of PE_i^2 by taking the greatest possible advantage of parallelism shown in column 7 of the third row of Table 1, where $\otimes$ and $\oplus$ denote the times required for performing a multiplication and an addition respectively. The time complexities for the computations required for PE_i^3, PE_i^4 and PE_i^5 can be obtained similarly to that of PE_i^2. However, the time complexity of PE_i^1 cannot be analyzed unless the function $p(y_i)$ is given.

2.6. *Summary of the operations of VLSI array processors.* We can summarize the operations of the VLSI array processors shown in Fig. 1 as follows. The computations starts from PE_i^1, which commands PE_i^5 to send the initial value of $y_i(l)$, and then computes $E_i(l)$ and $p_{y_i}(l)$. $E_i(l)$ is sent to PE_i^3, while $p_{y_i}(l)$ is sent to PE_i^2. After receiving $p_{y_i}(l)$, PE_i^2 commands PE_i^4 to send the initial value of λ, and then calculates $\hat{d}y_i$, which is sent to PE_i^3 and PE_{i-1}^3. After receiving $\hat{d}y_i$ and $\hat{d}y_{i+1}$, PE_i^3 will compute $\nabla_{\lambda_i}\phi(\lambda(t))$ and $\nabla^2_{\lambda_i\lambda_i}\phi^u(\lambda(t))$, which are sent to PE_i^4. PE_i^4 will then compute $\lambda_i(t+1)$ and send to PE_i^2 and PE_{i+1}^2, provided that $t < t_{max}$. The PE arrays formed by the PE_i^2 array, PE_i^3 array and PE_i^4 array will perform the Lagrange-dual Jacobi method until $t = t_{max}$ is detected in PE_i^4. When $t = t_{max}$, the PE_i^4 array will command the PE_i^2 array to send the data $\hat{d}y_i(\lambda(t))$ to the PE_i^5 array. Then the PE_i^5 array will update $y_i(l+1)$ and continue the above process until $l = l_{max}$ is detected in PE_i^5 and halt the execution.

3. *Application to receding-horizon controller*

3.1. *The implementable receding-horizon controller.* For a nonlinear system with control constraints described by $\dot{x} = f(x(t), u(t))$, $u(t) \in \Omega$, where $f: \mathbb{R}^k \times \mathbb{R}^p \to \mathbb{R}^k$ is twice continuously differentiable and satisfies $f(0,0) = 0$ and Ω is the set of admissible controls containing a non-empty convex polytope, Mayne and Michalska (1990a, b, 1991) proposed a globally stable implementable receding-horizon controller. Their control strategy employed a hybrid system $\dot{x}(t) = f(x(t), u(t))$, when $x(t) \notin W$, $\dot{x}(t) = Ax(t)$ otherwise, where $A = f_x(0,0) + f_u(0,0)$ is formed by applying a linear feedback control $u = Cx$ to the linearized system in a neighborhood W with small enough radius and centered at origin, where C is the feedback gain matrix. Their algorithm first calculates an admissible control and horizon pair

$$[u_0, t_{f,0}] \in Z_W(x_0), \tag{14}$$

where the initial state $x_0 \notin W$ is assumed, the set $Z_W(x) \equiv \{u \in S, t_f \in (0, \infty) \mid x^u(t + t_f; x, t) \in \delta W\}$, where S denotes the set of all piecewise-continuous functions, x^u denotes the resulting state after applying the control u and δW denotes the boundary of W. The algorithm then sets $h = 0$, $t_h = 0$, $u_h = u_0$, $t_{f,h} = t_{f,0}$ and performs the following process repeatedly to yield the receding-horizon feedback control.

It applies the obtained control u_h for $x \notin W$ and/or the linear feedback control Cx for $x \in W$ to the real system over $[t_h, t_h + \Delta t]$, where $\Delta t \in (0, \infty)$. Let x_{h+1} be the resulting state at $t_{h+1} (= t_h + \Delta t)$; then, if $x_{h+1} \in W$, the algorithm switches the control to $u = Cx$ over (t_{i+1}, ∞); otherwise, it calculates an improved control and horizon $[u_{h+1}, t_{f,h+1}]$ in the sense that

$$\begin{aligned} &[u_{h+1}, t_{f,h+1}] \in Z_W(x_{h+1}), \\ &V(x_{h+1}, t_{h+1}, t_{f,h+1}) \le V(x_{h+1}, t_{h+1}, u_h, t_{f,h} - \Delta t) \end{aligned} \tag{15}$$

where

$$V(x, t, u, t_f) \equiv \int_t^{t+t_f} \tfrac{1}{2}(\|x^u(\tau; x, t)\|_Q^2 + \|u(\tau)\|_R^2)\, d\tau + \int_{t+t_f}^{\infty} \tfrac{1}{2}(\|x_L(s; x^u(t+t_f; x, t)\|_Q^2)\, ds,$$

in which R and Q are positive-definite matrices, $\|y\|_A^2$ denotes $y^T Ay$, and x_L denotes the state trajectory in region W with feedback control $u = Cx$.

The two-phase parallel computing method (Lin, 1994) aims to obtain a receding-horizon feedback control solution for every Δt time interval $\hat{u}(\tau)$, $t \le \tau \le t + \Delta t$, based on Mayne and Michalska's algorithm. In the first phase, we discretize the system into N time intervals and use slack variables to formulate the following *feasibility problem*, which can also be called the phase 1 problem, to obtain an admissible control and horizon pair as required in (14):

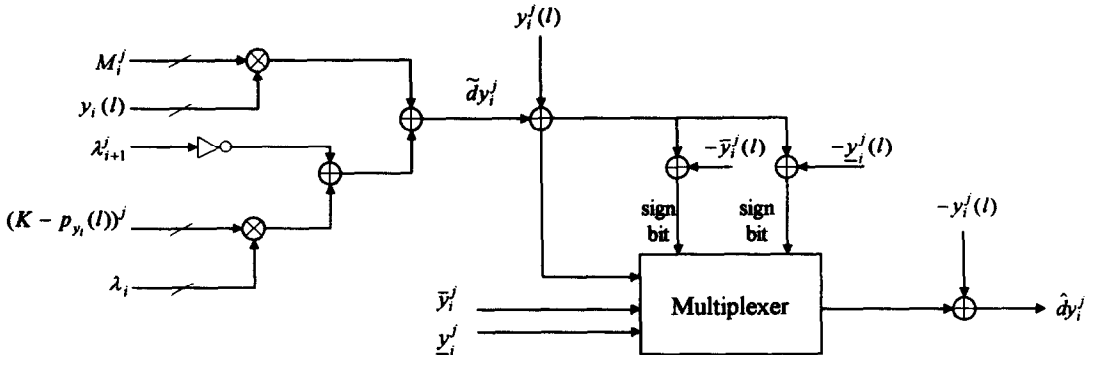
$$\min \sum_{i=0}^{N} s_i^T s_i, \tag{16a}$$

$$x_{i+1} - x_i - \frac{t_f}{N} f(x_i, u_i) + s_i = 0, \quad x_0 = x(t), \tag{16b}$$

$$x_N + s_N = 0, \tag{16c}$$

$$au_i + b \le 0, \quad i = 0, \ldots, N-1, \tag{16d'}$$

in which s denotes the vector of slack variables, and we explicitly express the non-empty convex polytope in Ω by the set of q-dimensional linear inequality constraints on $au + b \ge 0$, where the matrix $a \in \mathbb{R}^{q \times p}$ and the vector $b \in \mathbb{R}^q$. To apply the algorithm we proposed in Section 2, we need to



LEGEND: $(\cdot)^j$ - the jth component if $(\cdot)$ is a vector
the jth row if $(\cdot)$ is a matrix

Fig. 3. The arithmetic unit for computing a component $\hat{d}y_i^j$ of the vector $\hat{d}y_i$ in PE_i^2.

reformulate the inequality constraints (16d'). First, we separate the simple inequality constraints $\underline{u}_i \le u_i \le \bar{u}_i$ from $au_i + b \ge 0$, and then convert the rest of the inequality constraints to equality constraints $a'u_i + b + z_i = 0$ by adding positive variables z_i, where $a' \in \mathbb{R}^{r\times p}$, $r \le q$. We can then rewrite (16d') as

$$\begin{aligned} &a'u_i + b + z_i = 0, \quad \underline{u}_i \le u_i \le \bar{u}_i, \\ &z_i \ge 0, \quad i = 0, \ldots, N-1. \end{aligned} \tag{16d}$$

If there is no simple inequality constraint for u_i, we may set $\bar{u}_i = \infty$ and $\underline{u}_i = -\infty$.

Suppose that the optimal objective value of the phase 1 problem (16a–d) under a proper horizon t_f is zero; this t_f and control solution is then the admissible horizon and control pair required in (14). Because t_f is unspecified in (16a–d), we use a two-level (master- and slave-level) approach to solve the phase 1 problem (16a–d). The program in the *master level* of the two-level method is to determine a t_f, which is passed to the *slave level*, and the *slave problem* is (16a–d) with a fixed t_f given by a *master program*. The *master program* is simple; it increases t_f by δt_f each iteration, where δt_f is a small positive real number. However, to increase the computational speed, we apply a gradient method for the first few iterations. Thus the master program (Lin, 1994) is as follows:

$$\begin{cases} t_f(l+1) = t_f(l) + \gamma \dfrac{\mathrm{d}}{\mathrm{d}t_f} \displaystyle\sum_{i=0}^{N} \hat{s}_i^{\mathrm{T}}(t_f)\hat{s}_i(t_f) & \text{if } \displaystyle\sum_{i=0}^{N} \hat{s}_i^{\mathrm{T}}(t_f)\hat{s}_i(t_f) > \varepsilon, \\ t_f(l+1) = t_f(l) + \delta t_f & \text{if } \displaystyle\sum_{i=0}^{N} \hat{s}_i^{\mathrm{T}}(t_f(l))\hat{s}_i(t_f(l)) < \varepsilon \text{ and } \neq 0, \\ \text{stop} & \text{otherwise,} \end{cases} \tag{17}$$

where $\hat{s}(t_f)$ denotes the solution of the slack variables in the slave problem under a given t_f. The *slave problem* is a case of the nonlinear programming problem considered in Section 2, which can be solved by the algorithm presented in Section 2.

When the master program stops, the zero objective value of the phase 1 problem is achieved. This means that the admissible control and horizon pair is obtained. Let $[\tilde{u}, \tilde{t}_f]$ denote the admissible control and horizon pair obtained from phase 1 of the two-phase method; the phase 2 method will then improve $[\tilde{u}, \tilde{t}_f]$ in the sense of reducing the performance index $V(x, t, u, t_f)$, as required in (15). Thus, in phase 2, we shall solve the following *phase 2 problem*:

$$\begin{aligned} &\min \sum_{i=0}^{N-1} (\tfrac{1}{2}x_i^{\mathrm{T}}Qx_i + \tfrac{1}{2}u_i^{\mathrm{T}}Ru_i), \\ &x_{i+1} - x_i - \frac{\tilde{t}_f}{N} f(x_i, u_i) = 0, \quad x_0 = x(t), \quad x_N = 0, \\ &a'u_i + b + z_i = 0, \quad \underline{u}_i \le u_i \le \bar{u}_i, \quad z_i \ge 0, \quad i = 0, \ldots, N-1, \end{aligned} \tag{18}$$

which is a discretized version of (15).

We see that the phase 2 problem is also a case of the nonlinear programming problem considered in Section 2, and so can be solved by the algorithm presented in Section 2.

3.2. *The VLSI-array-processor architecture and operations for the implementable receding-horizon controller.* The slave problem (16a–d) with a fixed t_f is a special case of (1) with appropriate dimensions. The phase 2 problem (18) is also a special case of (1) with

$$y_i = (x_i, u_i, s_i, z_i), \quad M_i = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & I & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$K = \begin{bmatrix} I & 0 & -I & 0 \\ 0 & -a'I & 0 & -I \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$p(y_i) = \begin{bmatrix} f(x_i, u_i) \\ b \\ 0 \\ 0 \end{bmatrix}, \quad \psi(y_N) = x_N + s_N,$$

where the 0s in M_i and K and the 0s in $p_i(y_i)$ denote respectively the zero submatrix and the zero vector with appropriate dimensions. The phase 2 problem (18) is also a special case of (1) with

$$y_i = (x_i, u_i, z_i), \quad M_i = \begin{bmatrix} Q & 0 & 0 \\ 0 & R & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad i \ne N,$$

$$M_N = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad K = \begin{bmatrix} I & 0 & 0 \\ 0 & -aI & -I \\ 0 & 0 & 0 \end{bmatrix}.$$

$$p(y_i) = \begin{bmatrix} f(x_i, u_i) \\ b \\ 0 \end{bmatrix}, \quad \psi(y_N) = x_N.$$

Thus the VLSI-array-processor architecture presented in Fig. 1 can readily be used to implement the solution methods for the solution methods for the slave problem and the phase 2 problem. Although the computing formulas for these two problems are the same, the data in these formulas are different. Therefore we need to use one bit to represent the mode for these two problems. We let 0 represent solving the slave problem and 1 represent solving the phase 2 problem. Thus this one-bit mode can be used to control a multiplexer to select the corresponding data, as shown in Fig. 4 for the calculation in PEs.

What remains for the architecture to implement in the two-phase parallel computing method is the master program in phase 1. As we have shown in (17), the master program is very simple. Thus we may use a processing element PE^8 to perform the formula given in (17). However, the data required in PE^8 should be provided from the solution of the slave problem from all time intervals. Let $(\hat{s}(t_f), \hat{x}(t_f), \hat{u}(t_f), \hat{\lambda}(t_f))$ be the solution of the slave problem under a given t_f; then we need PE^6 array processors to calculate the values of $\hat{s}_i^{\mathrm{T}}(t_f)\hat{s}_i(t_f)$ and $-\hat{\lambda}_i^{\mathrm{T}}(t_f)(\partial/\partial t_f)f(\hat{x}_i(t_f), \hat{u}(t_f))$, and pyramid-like $\log_2(N+1)$ stage PE^7 array processors. The PE^7 array processors are two-input adders in the upward direction used to form the sums $\sum_{i=0}^{N} \hat{s}_i^{\mathrm{T}}(t_f)\hat{s}_i(t_f)$ and $\sum_{i=0}^{N} \hat{\lambda}_i^{\mathrm{T}}(t_f)(\partial/\partial t_f)f(\hat{x}_i(t_f), \hat{u}_i(t_f))$, which equals $(\mathrm{d}/\mathrm{d}t_f)\sum_{i=0}^{N} \hat{s}_i^{\mathrm{T}}(t_f)\hat{s}_i(t_f)$ (Lin, 1994). These are the data needed in PE^8 to perform the gradient method when $\sum_{i=0}^{N} \hat{s}_i^{\mathrm{T}}(t_f)\hat{s}_i(t_f) < \varepsilon$. The PE^7 array processors are registers in the downward direction used to propagate the value of t_f computed in the *master program* to the *slave problem*. Thus the overall VLSI array processors to implement the two-phase method are as shown in Fig. 5, in which the architectures of PE_i^1, PE_i^2, PE_i^3, PE_i^4 and PE_i^5 arrays are almost the same as in Fig. 1, except for the addition of dotted links directed from PE_i^5 to PE_i^4, as explained below. Because, when the slave problem is solved for a given t_f, the data $\hat{\lambda}(t_f)$ stored in PE_i^4 should be sent to PE_i^6; these dotted links represent the fact that when PE_i^5 detects convergence of the slave problem, it will command PE_i^4 to send data to PE_i^6. Therefore there also exist solid links from PE_i^4 to PE_i^6 in Fig. 5. Note that in phase 1, PE_i^5 will not halt the execution when detecting $l = l_{\max}$. This is different from Fig. 1.

3.3. *The operations of the VLSI array processors for the two-phase method.* Initially, PE^8 will provide a value of $t_f(0)$ and set the mode to be 0, and will pass down the value of

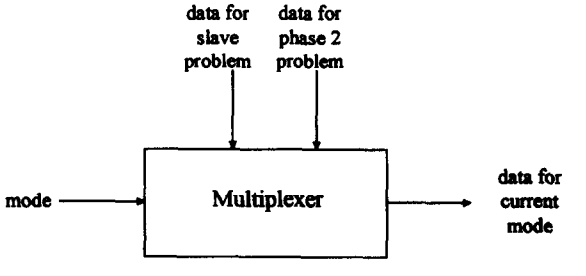


Fig. 4. The multiplexer controlled by the problem mode for selecting the corresponding data.

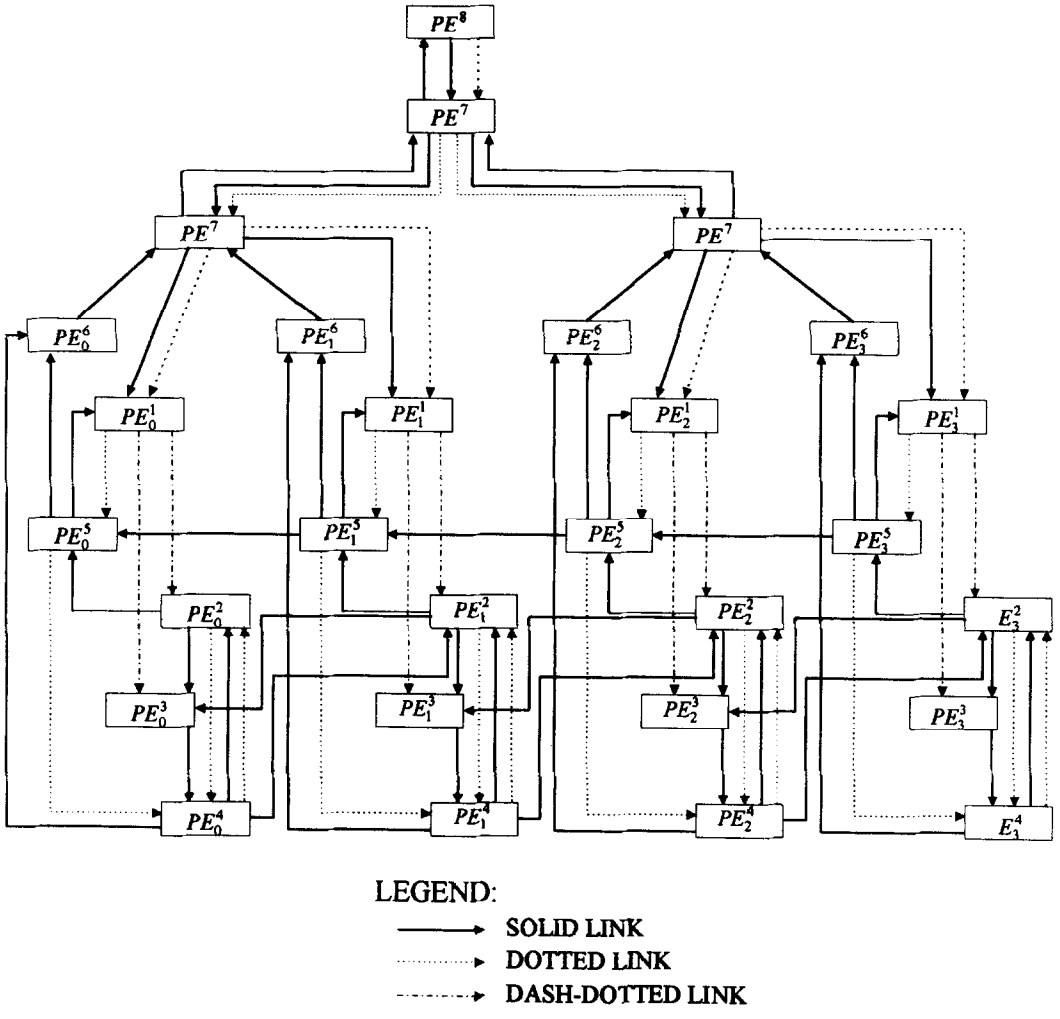


Fig. 5. VLSI-array-processor architecture for the two-phase parallel computing method with $N = 3$.

$t_f(0)$ and mode 0 to the PE_i^1 array processors through the PE^7 pyramid-like array processors, as shown by the solid links directed from PE^8 through the PE^7 arrays to the PE_i^1 array in Fig. 5. Then the PE_i^1, PE_i^2, PE_i^3, PE_i^4 and PE_i^5 arrays will perform the algorithm proposed in Section 2 to solve the slave problem under the value of t_f given by PE^8 until convergence in PE_i^5 is detected, that is, when $l = l_{max}$. The PE_i^5 array processors will then output the values of $\hat{s}_i(t_f)$ and $(\partial/\partial t_f)f(\hat{x}_i, \hat{u}_i)$ to the PE_i^6 array processors, and will command the PE_i^4 array to send the data of $\hat{\lambda}_i(t_f)$ to PE_i^6. The PE_i^6 will compute $\hat{s}_i(t_f)\hat{s}_i(t_f)$ and $\hat{\lambda}_i^T(t_f)(\partial/\partial t_f)$ $f(\hat{x}_i(t_f), \hat{u}_i(t_f))$, and the PE^7 arrays of processors will perform the sums $\sum_{i=0}^{N} \hat{s}_i^T(t_f)\hat{s}_i(t_f)$ and $\sum_{i=0}^{N} \hat{\lambda}_i^T(t_f(\partial/\partial t_f)f(\hat{x}_i(t_f),$ $\hat{u}_i(t_f))$ $(=(d/d_{t_f}) \sum_{i=0}^{N} \hat{s}_i^T(t_f)$, $\hat{s}_i(t_f)$, and input them to the PE^8 processor to perform (17). This process will continue until PE^8 detects the convergence of the phase 1 problem, that is, $\sum_{i=0}^{N} \hat{s}_i^T(t_f)\hat{s}_i(t_f) = 0$; then the value of the admissible horizon t_f and the command of mode changing to 1 will be passed to the PE_i^1 array processors through the PE^7 arrays. The mode-change command is indicated by dotted links directed from PE^8 through the PE^7 arrays to the PE_i^1 array as shown in Fig. 5. The PE_i^1 array will then command the PE_i^2, PE_i^3, PE_i^4 and PE_i^5 arrays to change the mode to 1. For clarity, we do not show in Fig. 5 the dotted links for the rest of the mode-change command that occurs among the PE_i^1, PE_i^2, PE_i^3, PE_i^4 and PE_i^5 arrays. At this point, the solution of the phase 1 method, $\hat{x}_i$, $\hat{u}_i$, $\hat{z}_i$, $i = 0, \ldots, N$, is stored in the PE_i^5 array. The PE_i^1, PE_i^2, PE_i^3, PE_i^4 and PE_i^5 arrays will then proceed to solve the phase 2 problem until $l = l_{max}$ is detected in PE_i^5, which will halt the execution and output the solution.

3.4. *Overall-time complexity.* From Section 3, we see that all the computations of the two-phase method lie in the Lagrange-dual Jacobi method; thus the total time complexity spent in the Lagrange-dual Jacobi method is the dominant term of the overall-time complexity. Let m_s denote the actual numbers of iterations that the iterative *two-level* phase 1 problem takes to converge. Then the total number of iterations of the *Lagrange-dual Jacobi method* performed in phase 1 is $m_s l_{max} t_{max}$. Furthermore, the total number of iterations of the *Lagrange-dual Jacobi method* performed in phas 2 is $l_{max} t_{max}$. The time complexity of the array PEs should count as only that of one PE, since they are executed asynchronously and simultaneously. Let T_{PE^j} denote the time complexity of PE^j, which is shown in column 7 of Table 1 in terms of numbers of $\otimes$ and $\oplus$. Also, let T_{CL} denote the time complexity of the asynchronous handshaking communication link, which is equal to 3 clock pulses according to the design in Kung (1988). Similarly, the time complexity of the array communication links should count as just one T_{CL}. Thus the total time complexity spent in the Lagrange-dual Jacobi method based on the above notation and the computing architecture shown in Fig. 5 is

$$(m_s l_{max} t_{max} + l_{max} t_{max})(T_{PE^2} + T_{PE^3} + T_{PE^4} + 3T_{CL}). \quad (19)$$

3.5. *Simulations.* According to the work of Yano *et al.* (1990), $T_{\otimes} = 3.8$ ns for a 16×16-bit multiplication, $T_{\oplus} \leq 0.2$ ns for an addition, and the period of a clock pulse is approximately 40 ps. We may calculate that $T_{PE^2} = [7.6 + 0.2 \log_2 (2n + 2)]$ ns, $T_{PE^3} = (7.6 + 0.2 \log_2 n^2)$ ns, and $T_{PE^4} = [7.6 + 0.2 \log_2 (n + 1)]$ ns, according to column 7 of Table 1, and $T_{CL} = 0.1$ ns. Then (19) becomes

$$(m_s l_{max} t_{max} + l_{max} t_{max}) \times \{23.1 + 0.2 \log_2 [(2n + 2 + 2)n^2(n + 1)]\} \text{ ns}. \quad (20)$$

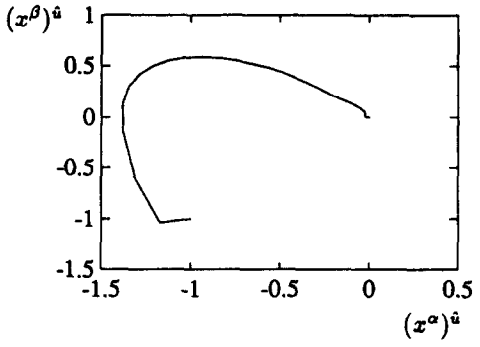


Fig. 6. The final complete state trajectory for the example.

Example. The Rayleigh equation.

$$\begin{aligned} \dot{x}^\alpha &= x^\beta, \quad x_0^\alpha = -1, \\ \dot{x}^\beta &= -x^\alpha + [1.4 - 0.14(x^\beta)^2] + 4u, \quad x_0^\beta = -1, \end{aligned} \tag{21}$$

where x^α and x^β are state variables and u is the scalar control. We intend to find a control solution that satisfies the instantaneous control constraints $|u| \le 0.7$ and that drives the system from the initial state $(-1, -1)$ at time $t = 0$ to $(0, 0)$ asymptotically. The following initial values are assumed in the phase 1 method: $t_f = 5$ s, $u_i = 0$, $0 \le i \le N-1$, and $x_i^\alpha = x_0^\alpha - (i/N)x_0^\alpha$, $x_i^\beta = x_0^\beta - (i/N)x_0^\beta$, $i = 0, 1, \ldots, N$. The matrix

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and $R = 1$ are used in phase 2. The linear feedback control $\hat{u} = -x^\alpha - 2x^\beta$ is employed in the region $W \equiv \{x \mid |x|_\infty \le 0.5\}$ to result in negative eigenvalues for the linearized closed-loop system at $(0, 0)$. The algorithmic parameters are arbitrarily assigned to be $N = 30$, $\varepsilon = 0.001$, $\gamma_1 = \gamma_2 = \gamma_3 = 0.1$, $\delta t_f = 0.2$ s, $l_{max} = 40$, $t_{max} = 40$. Solving the example by our two-phase method-based implementable receding-horizont feedback control algorithm, we obtain $h = 30$ before reaching the region W, and the final complete state trajectory is shown in Fig. 6.

Estimated computation time for the two-phase method. In this example, $n = 3$ which is composed of two states and one control. For all h, including $h = 0$, we have $m_s = 1$. The estimated computation time of the two-phase algorithm calculated from (19) is 0.08 ms. This shows that the receding-horizon controller hardware meets the real-time processing system needs.

4. *Conclusions*

We have presented the architecture of a basic hardware module to implement a nonlinear programming algorithm that solves discrete-time optimal control problems for nonlinear systems with quadratic objective function and control constraints. We have applied this basic hardware module in the two-phase method, and it results in a simpler algorithm than that in Lin (1994) for solving a receding-horizon feedback control solution. We have also presented the VLSI-array-processor architecture for this receding-horizon controller. The estimated computation time to obtain a receding-horizon feedback control solution is of the order of 0.1 ms, which meets the real-time processing requirement.

Acknowledgement—This research is supported by National Science Council, Taiwan under Grant NSC84-2213-E-009-132.

References

Bazaraa, M. and Shetty, C. (1979) *Nonlinear Programming*, Wiley, New York.

Clarke, D. W. and Scattolini, R. (1991) Constrained receding horizon predictive control. *Proc. IEE, Pt D* **138**, 347–354.

DeNicolao, G. and Scattolini, R. (1994) Stability and output terminal constraints in predictive control. In *Advances in Model-Based Predictive Control*, ed. D. Clarke, pp. 105–121. Oxford University Press.

DeNicolao, G., Magni, L. and Scattolini, R. (1996) On the robustness of receding-horizon control with terminal constraints. *IEEE Trans. Autom. Control* **AC-41**, 451–453.

Dyer, P. and McReynolds, S. (1970) *The Computation and Theory of Optimal Control*. Academic Press, New York.

Frantzeskakis, E. and Liu, K. (1994) A class of square root and division free algorithms and architecture for QRD-based adaptive signal processing. *IEEE Trans. Sig. Process.* **SP-42**, 2455–2469.

Kung, S. Y. (1988) *VLSI Array Processors*. Prentice-Hall, Englewood Cliffs, NJ.

Lin, S.-Y. (1993) A two-phase parallel computing algorithm for the solution of implementable receding horizon control for constrained nonlinear systems. In *Proc. 32nd IEEE Conf. on Decision and Control.*, San Antonio, TX, pp. 1304–1309.

Lin, S.-Y. (1994) A hardware implementable receding horizon controller for constrained nonlinear systems. *IEEE Trans. Autom. Control* **AC-39**, 1893–1899.

Luenberger, D. (1984) *Linear and Nonlinear Programming*. Addison-Wesley, Reading, MA.

Mayne, D. Q. and Michalska, H. (1990a) Receding horizon control of nonlinear systems. *IEEE Trans. Autom. Control* **AC-35**, 814–824.

Mayne, D. Q. and Michalska, H. (1990b) An implementable receding horizon controller for stabilization of nonlinear systems. In *Proc. 29th IEEE Conf. on Decision and Control*, Honolulu, HI, pp. 3396–3397.

Mayne, D. Q. and Michalska, H. (1991) Robust receding horizon control. In *Proc. 30th IEEE Conf. on Decision and Control*, Brighton, UK, pp. 64–69.

Mayne, D. Q. and Polak, E. (1993) Optimization based design and control. In *Proc. 12th IFAC World Congress*, Sydney, Australia, Vol. III, pp. 129–138.

Mosca, E. and Zhang, J. (1992) Stable redesign of predictive control. *Automatica* **28**, 1229–1233.

Richalet, J. (1993) Industrial applications of model based predictive control. *Automatica* **29**, 1251–1274.

Yano, K., Yamanaka, T., Nishida, T., Saito, M., Shimohigashi, K. and Shimizu, A. (1990) A 3.8-ns CMOS 16 × 16-b multiplier using complementary pass-transition logic. *IEEE J. Solid State Circ.* **25**, 388–395.