Int. J. Engng Ed. Vol. 24, No. 6, pp. 1170–1179, 2008
Printed in Great Britain.

0949-149X/91 $3.00+0.00

New Control Laboratory Using Parallel Programming*

S. DORMIDO-CANTO, J. SÁNCHEZ and S. DORMIDO
Department of Computer Science and Automatic Control, UNED. C/ Juan del Rosal 16, 28040, Madrid, Spain. E-mail:sebas@dia.uned.es

This paper discusses the viability of using parallel processing methods to solve control algorithms in real time in the field of Control Engineering education. It is a well-known fact that some types of control problems cannot be dealt with in just one practical session in the lab because of their huge computational load. However, the use of low-cost clusters of workstations (COWs) and message-passing software allows students to program their own control algorithms and visualize the results in real time without having to wait a long time. In this paper we analyze the control of a pH-neutralization process and the parallel performance of the algorithms proposed using an illustrative example, paying special attention to the speedup factor. Thus, this heavy-computational-load example gives a meaningful case study to demonstrate the suitability of using parallel computing techniques to include new experiments in the control lab.

Keywords: parallel computing; distributed computing; dynamic programming; laboratory optimal control; clusters of workstations; real time

MOTIVATION AND INTRODUCTION

SINCE MODERN CONTROL THEORY EMERGED, optimization methods have been a constant in the theoretical contents of Control Engineering graduate courses. Of these methods, the most relevant is dynamic programming (DP) [1], as it is a classical and powerful technique to solve several optimization problems under general conditions. Its many applications are well known [2, 3]: scheduling, automatic control, artificial intelligence, economics, etc.

Although this technique is a common factor in every theoretical course on optimal control, it is a resource that has not been widely used in control lab assignments because calculating the function cost is a very time-consuming task. Although dynamic programming can be applied analytically in some cases, generally the solution has to be found numerically and, now, unfortunately, the problem of dimensionality plays a key role: CPU time and memory storage requirements can become so high that, in practice, conventional dynamic programming cannot be used numerically at all, except to work out simple problems. For this reason, several techniques have been developed to reduce the computational cost [2, 4–11]. These techniques decrease the main disadvantage of DP, i.e. its great computational cost, but they do not solve it completely. The computational time is still very high in most cases of practical interest either in industrial or educational contexts, as for example, in control laboratory assignments.

One of the solutions to take advantage of dynamic programming in actual control problems, in other words to compute the control algorithm in one sampling interval, is a parallel machine. Since there are a large number of arithmetic operations that can evaluate parallely when the dynamic programming recursive formula is calculated, the use of parallel programming techniques will make it possible to reduce the execution time in order to solve large-scale dynamic programming problems. The computational theory of dynamic programming from the viewpoint of parallel computation was examined by Larson and Tse [12], but the resulting algorithms are only applicable to a highly specific and expensive range of parallel computer architectures.

However, the high price of parallel computers means that university departments cannot seriously consider this solution to introducing dynamic programming in control lab assignments. However, in recent years, the falling prices and technological advances in personal computers have made it possible to carry out parallel processing in a simple and not-so-expensive fashion by building clusters of workstations (COWs) [13–15]. Nowadays, COWs are considered a good low-cost alternative to parallel computers for many reasons (flexibility, scalability, and adaptability) but, in an educational context, it is the economic cost that stands out, especially that of the hardware and software. Just by hooking together a few Intel/AMD boxes with a dedicated Fast-Ethernet switch and installing any Linux distribution, a COW will be ready to crunch numbers. Once the machine is built, the last step is to choose the most convenient parallel programming paradigm, which in clusters is usually by passing messages among processing

* Accepted 17 July 2008.

nodes. There are many proprietary and public domain message-passing systems (CMMD, Express, Fortran-M, Nx, PARMACS, etc.), but the most important and popular packages are MPI (Message-Passing Interface) [16] and PVM (Parallel Virtual Machine) [17]. In brief, MPI is a standard specification developed by the MPI Forum, a consortium of parallel computer vendors, and PVM is a self-contained system to run parallel applications on a network of heterogeneous Linux/Windows computers.

Thus, it is clear that an infrastructure for parallel processing aimed at solving optimal control problems in real time is affordable at low cost for many university departments. Accordingly, it will be possible to include new, challenging control projects in traditional laboratory assignments. Up to now, the analysis, design, and construction of complex real-time control systems using dynamic programming algorithms in the lab has been a far-fetched idea. These types of projects have been prohibitive because of the time necessary to run the experiments when a complex process was being controlled. Now, low-cost parallel computers allow departments to extend the range of processes to be controlled in the lab, the time constraint being just a matter of scalability and adaptability: bringing down the sampling interval to evaluate the control algorithm can be obtained with a cluster re-size.

Currently, one of the pedagogical goals of our Department is to show students how parallel computating may be applied to solve many types of engineering problems. For this reason, all the UNED's computer science students [18] attend a course on advanced computer architecture where the principles of parallel computing are explained. There are many works on the contents and scheduling of these courses [19–22] that highlight that engineering undergraduates have to be conversant with the tools that parallel processing offer to solve certain problems. Thus, an understanding of parallel programming is fundamental to understanding the performance that can be obtained in the design and analysis of a broad range of control systems. In the quest for a more useful and pragmatic rather than theoretical view of parallel computing, the core of the course focuses on teaching how to construct COWs and program them using the message-passing paradigm. Therefore, once students have passed the course, they should be able to build and program a COW using low-cost facilities, i.e., Linux as an operating system and PVM as a message-passing library.

Other students and pedagogical scenarios for which the control laboratory described in this paper can be appropriate are chemical engineering and environmental science students. In both instances, although for different reasons, these students must know about the pH-neutralization process of liquids. Unfortunately, their knowledge of parallel computing and programming techniques is not as extensive as that of computer science graduates, but their control knowledge can be good enough in most cases to understand and manage the pH-neutralization process of a liquid stored in a tank. Accordingly, the application of this lab for control courses for chemistry and environmental students is possible, but such issues as the speedup factor and parallel performance are omitted. This means that students would use the lab from a pure point of view, i.e., just taking into consideration the tuning of predictive controllers in order to obtain a system response that fulfills some design specifications. Nevertheless, the application of these parallel-computing techniques should not be totally ruled out for doctoral courses since they will provide chemical engineers with a good background to do research on new control strategies to be applied to different chemical industry processes.

This paper demonstrates how pH process modeling and regulator design can be integrated into a cluster of PCs, producing a new category of control experiments to be developed in labs. It is further demonstrated that the experiments can be implemented in real time.

The paper is organized as follows: the next section briefly introduces classical dynamic programming algorithms. This is followed by sections describing: the parallel implementations of these algorithms in COWs using the message-passing paradigm; the main features of the cluster and software used to program the previous algorithms; the control of a pH-neutralization process via a cluster as a new control lab assignment; and the scalability of the previous control problem and the viability of implementing it in real time in the lab using a cluster and an improved parallel version of a classical dynamic programming algorithm. Finally, the contributions of this work are summarized.

CLASSICAL DYNAMIC PROGRAMMING ALGORITHMS

Dynamic programming (DP) is based on Bellman's Principle of Optimality [1]. Basically, it states that every portion of an optimal trajectory is an optimal trajectory for a particular subproblem as depicted in Fig. 1.

Bellman's Principle of Optimality: if I + II + III is the trajectory from state x_A to state x_B, according to a given cost (performance) function, then II is the optimal trajectory for the subproblem x_C–x_D.

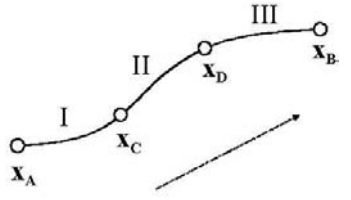

Fig. 1. Bellman's Principle of Optimality.

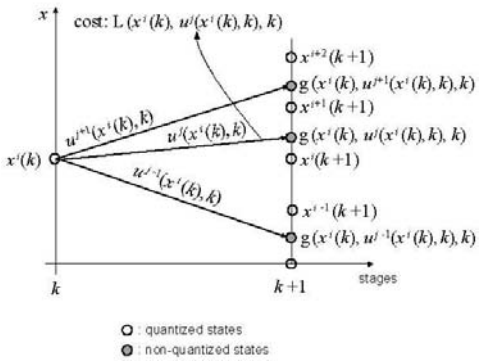

Fig. 2. Basic backward dynamic programming computational procedure at a given state $x^i(k)$.

In this problem, x is the *state variable*, X is the set of allowable states, u is the *decision variable*, U is the set of admissible decisions, k is the *stage*, and J is the cost or objective function; L represents the cost of a single stage.

If the minimum cost function from stage k to the end of the decision problem is defined as:

$$I(x,\ k) = \min_{u(k),\, u(k+1),\, \ldots,\, u(N)} \left\{ \sum_{j=k}^{N} L(x(j),\ u(j),\ j) \right\}$$

it is possible to prove using Bellman's Principle of Optimality that

$$I(x,\ k) = \min_{u} \{ L(x(k),\ u(k),\ k) + I\,[g(x(k),\ u(k),\ k),\ k+1] \} \quad (1)$$

with

$$I(x,\ k) = \min_{u(N)} \{\ L(x(N),\ u(N),\ N)\ \}$$

for the final stage N.

In order to solve (1) numerically, the sets X and U are assumed to be finite for computational purposes; in those instances where they are infinite, the set of admissible states X and the set of admissible decisions U are quantized at each stage and a computational grid defined:

$$X(k) = \{x^1(k),\ x^2(k),\ .....,\ x^{M_X(k)}(k)\}$$

$$U(x(k),\ k) = \{u^1(x(k),k),\ u^2(x(k),k),\ .....,\ u^{M_U(x(k),\,k)}(x(k),k)\}$$

where $M_X(k)$ is the number of quantized states at stage k and $M_U(x(k), k)$ is the number of quantized decisions at stage k and state $x(k)$.

The computational method usually proceeds backwardly (*backward dynamic programming with interpolation*), as shown in Figs 2 and 3 (Madrid *et al.* 1996). $u^*(x^i(k),k)$ stands for the optimal decision at state x^i at stage k. The optimal decision policy is obtained for a complete family of optimization problems, i.e., for every state at all stages, and it always determines an absolute minimum within the accuracy of the computational grid as shown in Fig. 4.

It must be taken into account that if $g\,(x^i(k),\ u^j(x^i(k),k),\ k)$ is not a quantized state, then $I(g\,(x^i(k),\ u^j(x^i(k),k),\ k),k+1)$ has to be interpolated.

It has been proven, under reasonable assumptions, that interpolation errors tend to increase almost linearly with $(N-k)$. The only way to be more accurate is to use more quantized states and decisions, with a higher computational load.

However if the inverse function g^{-1} exists,

$$g\big(x(k),\ g^{-1}[\,x(k+1),\ x(k)\,],\ k\big) = x(k+1)$$

an alternative sequential backward dynamic programming computational procedure without interpolation can be used (Fig. 5) [9]. As there are no errors due to interpolation, it is clear that the only way to obtain a more accurate solution is to use a dense computational grid.

initialize $I(x,\ k) = \infty, \quad \forall x \in X(k),\ \forall k$
evaluate $I(x,\ N), \quad \forall x \in X(N)$
for all the stages from $k = N-1$ to 1
 for all the quantized states $x^i(k) \in X(k)$
 for all the admissible controls $u^j(k) \in U(x^i(k),\ k)$
 evaluate $g(x^i(k),\ u^j(k),\ k)$
 if $g(x^i(k),\ u^j(k),\ k) \in X(k+1)$
 interpolate $I(g(x^i(k),\ u^j(k),\ k),\ k+1)$
 if $L(x^i(k),\ u^j(k),\ k) + I(\ g(x^i(k),\ u^j(k),\ k), k+1) < I(x^i(k),\ k)$
 $I(x^i(k),\ k) = L(x^i(k),\ u^j(k),\ k) + I(\ g(x^i(k),\ u^j(k),\ k),\ k+1)$
 $u^*(x^i(k),\ k) = u^j(k)$
 endif;
 endif;
 endfor;
 endfor;
endfor

Fig. 3. Sequential backward dynamic programming algorithm with interpolation.

The solution to (1) is by far the most time-consuming part of the dynamic programming computations. The approximate computation time τ, assuming there are no constraints, is:

$$\tau = \sum_{k=1}^{n} M_X(k) \cdot M_U(x(k), k) \cdot \Delta\tau$$

where $\Delta\tau$ is the time to solve (1) once, i.e. at one state using one decision choice. If there were constraints, (1) would have to be solved fewer times and the actual value of τ would be smaller.

Yet any increase in the number of states and decisions produces a fast growth of the computing time. Consequently, in order to solve many optimization problems with DP it will be necessary to resort to parallel processing. The parallel computation schemes will be discussed in the following section.

PARALLEL DYNAMIC PROGRAMMING ALGORITHMS

To parallelize the dynamic programming algorithms effectively, we need to know which stages are computation intensive and can be subdivided to parallelize them. First, it must be noted that the evaluation of the optimal return function, equation (1), for all stages generally involves three nested iterative loops. The internal loop varies depending on algorithms with or without interpolation, as described in Figs 3 and 5. Several approaches to parallelize the dynamic programming algorithms are possible [23].

In the following paragraphs, dynamic programming parallel procedures implemented on clusters using message passing are proposed to solve optimal control problems. The master/slave paradigm has been used as a programming paradigm to develop the parallel algorithms. The master is responsible for dividing the problem into small tasks, distributing these tasks between a farm of slave processors and gathering the partial results to produce the overall result. The slave processors execute a very simple code: receive a message with data, process the information, and send the result to the master. The work is done in stages: each stage must finish before the work for the next stage can be generated. Thus, the master synchronizes the slaves at the end of each stage. In the following sections the classical dynamic programming algorithms —with and without interpolation— are parallelized.

Table 1 summarizes the notations and conventions used throughout the rest of the text.

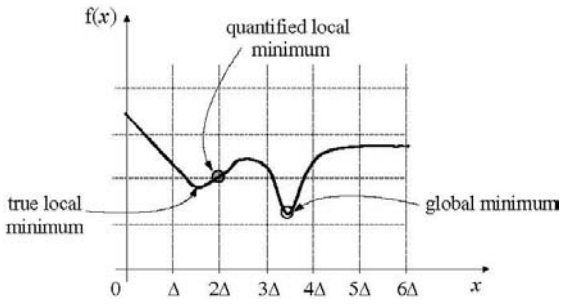

Fig. 4. A global minimum can be lost when a function is evaluated in too coarse a computational grid.

initialize $I(x, k) = \infty, \quad \forall x \in X(k), \ \forall k$
evaluate $I(x, N), \quad \forall x \in X(N)$
for all the stages from $k = N-1$ to 1
 for all the quantized states $x^i(k) \in X(k)$
 for all the quantized states $x^j(k+1) \in X(k+1)$
 $u = g^{-1}(x^j(k+1), x^i(k))$
 if $u \in U(x(k), k)$
 if $L(x^i(k), u(k), k) + I(x^j(k+1), k+1) < I(x^i(k), k)$
 $I(x^i(k), k) = L(x^i(k), u(k), k) + I(x^j(k+1), k+1)$
 $u^*(x^i(k), k) = u^j(k)$
 endif;
 endif;
 endfor;
 endfor;
endfor

Fig. 5. Sequential backward dynamic programming algorithm without interpolation.

Parallel algorithms without interpolation

Table 1. Notation and conventions

Notation	Meaning
k	index of stage
m	index of processor
M	number of slave processors
N	number of stages
Δx	partition size in the space of states
Δu	partition size in the space of decisions
$(\cdot)^*$	optimal value of $(\cdot)$
$(\cdot)_i$	i-th component of vector $(\cdot)$
$(\cdot)^i$	i-th quantized value of $(\cdot)$
$[(\cdot)^i]^m$	i-th quantized value of $(\cdot)$ computed by the processor m
$[(\cdot)]^m$	quantized values of $(\cdot)$ computed by the processor m
$[(\cdot)]^m_{start}$	initial value in the processor m of the quantized values of $(\cdot)$
$[(\cdot)]^m_{end}$	final value in the processor m of the quantized values of $(\cdot)$

MASTER
start up the parallel virtual machine: pvm_start_pvmd( );
start up the slave tasks: pvm_spawn( );
initialize $I(x, k) = \infty \ \forall x \in X(k) \ \forall k$
evaluate $I(x, N) \ \forall x \in X(N)$
send constant data to all slave processors: pvm_mcast( );
for $k = N-1$ to 1
 for $m = 1$ to M
 compute $[x(k)]^m_{start}$, $[x(k)]^m_{end}$
 send to each slave processor: $I(x, k), u(x, k) \quad \forall x \ \forall k$,
 ... $[x(k)]^m_{start}$, $[x(k)]^m_{end}$: pvm_send( );
 endfor
 receive the result from each slave processor:
 $I([x]^m, k), u^*([x]^m, k)$: pvm_recv( );
 compute and update $I(x, k), u(x, k) \quad \forall x \ \forall k$
endfor

Fig. 6. Master procedure for backward dynamic programming algorithm without interpolation.

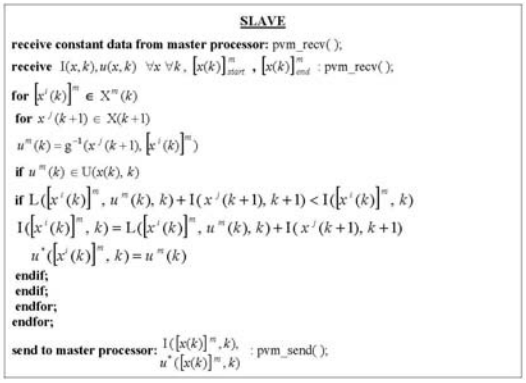

Fig. 7. Slave procedure for backward dynamic programming algorithm without interpolation.

In sequential dynamic programming without interpolation (Fig. 5), the decision variables are not quantized. However, when the decision variables can take any value for any quantized state at the current stage, the state at the next stage is also a quantized state. For this reason, the computational grid is just defined in the set X. When this algorithm is parallelized, the parallel processing can only be carried out in the loop of the states at stage k. The pseudocode corresponding to the master and slave processors is shown in Figs 6 and 7, respectively.

Parallel algorithms with interpolation

In sequential dynamic programming with interpolation it is necessary to define a quantized computational grid in the sets X and U (Fig. 3). The parallel processing can be carried out either in the loop of the states at stage k, or in the loop of the decisions at stage k. In both instances, an interpolation procedure to compute Equation (1) has to be used. Both parallel codes can be found in [24].

The parallel processing algorithm of the states makes use of the parallel processing carried out in the loop of the states at stage k. Each slave processor initially receives a subset of quantized states at stage k from the master. Every single quantized decision permitted has to be checked for every quantized state. Yet, in the parallel processing of the decisions the optimization procedure is carried out in two parts at each stage k. In the first part, each slave processor receives just a subset of the admissible decisions from the master and subsequently performs the optimization over all the quantized states at stage k using this subset of decisions. Thus each slave processor obtains a local optimum that is sent to the master. In the second part of the algorithm, the master, once all the local optima have been gathered, computes the actual global optimum.

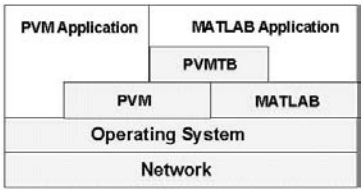

Fig. 8. Overview of PVMTB

CLUSTER AND SOFTWARE DESCRIPTION

The cluster used in this study consists of 16 AMD K7 processors (nodes) running at 500 MHz, each one with 384 MB of RAM and a 7 GB disk. The nodes are connected to a Fast Ethernet network via a 100 Mb/s switch, making up a COW with 1 master and 15 slave processors. The operating system installed is Linux (Red-Hat 6.1). This COW is isolated from any external network, and is exclusively for solving the optimization problem.

For this work, a parallel processing toolbox developed in Matlab was used [25]: PVMTB (Parallel Virtual Machine ToolBox), based on the standard PVM. With PVMTB, users of scientific computing environments, like Matlab, in a COW with a message-passing system, like PVM, can now take advantage of the rapid prototyping nature of the environment and the clustered computing power in order to prototype High Performance Computing (HPC) applications. The user maintains all the interactive, debugging and graphic capabilities, and can now reduce execution time by taking advantage of the processors available. The interactive capability can be regarded as a powerful, didactical and debugging tool.

Figure 8 shows a diagram of PVMTB. The Toolbox makes use of the PVM low-level routines and the Matlab-API (Application Program Interface) functions allow the exchange of messages between Matlab processes.

A CASE STUDY: THE CONTROL OF A PH-NEUTRALIZATION PROCESS

The pH process is of great importance in the chemical industry and in waste water treatment, and it is difficult to control for a number of reasons: (1) the process is highly nonlinear; (2) it is very sensitive to disturbances near the point of neutrality; (3) it is difficult to formulate and identify a mathematical model of the process due to small amounts of polluting elements, e.g. carbonate or phosphate, which change the process dynamics.

The experimental process

The experimental process consists, as Fig. 9 shows, of neutralizing a strong acid (HCl) with strong base (NaOH) in a continuous stirred tank reactor (*cstr*) with volume (V). The acid flow (q), whose concentration is c_A (mol/l), is adjusted manually and the base flow (u), whose concentration is c_B (mol/l), is controlled by a low-flow pneumatic valve, which is regulated with a predic-

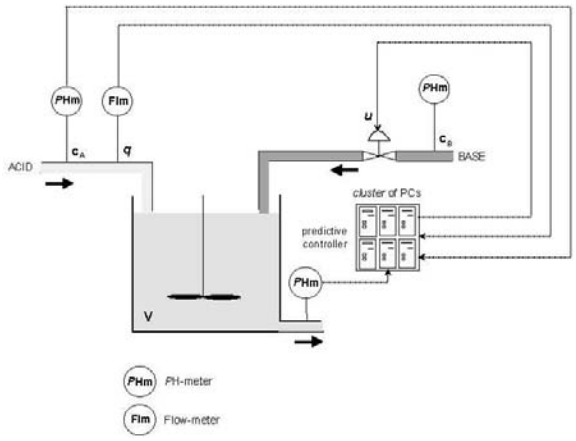

Fig. 9. *p*H neutralization of strong acid (concentration c_A, flow q) with strong base (concentration c_B, flow u). Tank volume is V.

Table 2. Experimental operating conditions

Acid flow, q	0.5 l min^{-1}
Base flow, u	0–0.1 l min^{-1}
Acid normality, c_A	$\approx 10^{-4}$ mol l^{-1}
Base normality, c_B	$\approx 0.5 \times 10^{-3}$ mol l^{-1}
Tank volume, V	10 l

tive controller implemented by a cluster of PCs. This feedback signal is used to provide a flow control loop at the cluster output so that this output could be regarded as the adjusting base flow, rather than the valve position. The *p*H level is measured in the outlet stream of the tank and sampled by the cluster.

Let x_A and x_B be the concentrations of acid and base in the tank respectively. The system dynamics is then given by

$$\begin{cases} \frac{dx_A}{dt} = \frac{q}{V}(c_A - x_A) \\ \frac{dx_B}{dt} = \frac{u}{V}c_B - \frac{q}{V}x_B \end{cases} \tag{2}$$

and the *p*H is given by

$$pH(x) = -\log\left(\sqrt{\frac{x^2}{4} + K_w} - \frac{x}{2}\right) \tag{3}$$

where $x = x_A - x_B$ and $K_w = 10^{-14}$ (mol/l)^2 at 25°C. The experimental operating conditions used in our case study are listed in Table 2.

The control system

The control purpose is to maintain the *p*H at a set point of the outlet stream by manipulating the base flow that reaches the tank at a rate determined by the position of a valve. Thus, the position of this valve is the control input that determines the neutralization in the tank, requiring continuous adjustment under feedback control in order to achieve satisfactory results. In this case study, the aim of the cluster of PCs is to replace a conventional PID controller.

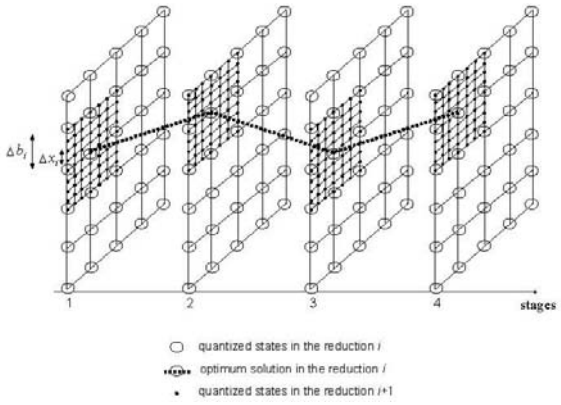

Fig. 10. Systematic reduction of computational grid without interpolation procedure with dimension 2.

So that students can work in parallel programming to solve control problems in real time, we developed a predictive controller based on dynamic programming using a cluster of PCs. The control parameters in our case study are $N_1 = 1$, $N_2 = 10$, $N_u = 1$ and $\lambda = 0$ (more details about predictive controllers can be found in [26]). Since predictive controllers make use of a process model to obtain the control signal by minimizing a given cost function, the controller is associated with an optimization problem with constraints, and it can thus be formulated as a dynamic programming problem. Thus, considering (2), (3) and sampling with Euler approximation where Δt is equal to T (sampling period), it is possible to write the state equation in terms of *p*H:

$$pH(k+1) = \frac{T\log e}{10^{pH(k)-14} + 10^{-pH(k)}} \cdot \left[-\frac{q}{V} \cdot \left(10^{pH(k)-14} - 10^{-pH(k)}\right) + \frac{c_B}{V}u(k) - \frac{c_A}{V}q \right] + pH(k)$$

EXPERIMENTAL RESULTS AND PERFORMANCE EVALUATION

To solve the problem in real time, students develop an improved parallel version of a parallel backward dynamic programming algorithm without interpolation known as *systematic reduction of computational grid without interpolation* [24] in the lab. In this new approach, students have to introduce a new external loop: *the number of reductions*. Consequently, they have to solve the dynamic programming problem as many times as the number of reductions. Once a solution is obtained for a reduction, a band of width $2\Delta b_i$ (i goes from 1 to the number of reductions) is calculated around

Table 3. Δx and Δb for three reductions

Reductions	Δx	Δb
Initial partition	0.5	8
1st reduction	0.25	1
2nd reduction	0.125	0.5
3rd reduction	0.0625	0.25
Initial partition	0.1	8
1st reduction	0.05	0.2
2nd reduction	0.025	0.1
3rd reduction	0.0125	0.05
Initial partition	0.01	8
1st reduction	0.005	0.02
2nd reduction	0.0025	0.01
3rd reduction	0.00125	0.005

it. Then a new computational grid with a lower Δx_i is computed for the next reduction. Accordingly, a better solution with a much lower computational complexity is obtained. Figure 10 depicts the procedure.

Numerous simulations are afforded in the lab using different sizes to define the initial quantized computational grids in the set X. Table 3 shows the partition size (Δx) and the width of the band (Δb) in the space of states when three reductions are considered.

Figure 11 shows some results for different set points in the *p*H control with $\Delta x = 0.5$ and $\Delta x = 1$ as initial partitions and an initial value of *p*H equal to 4.

Table 4 shows the average times for each sampling time with different sizes of the initial partition in a single processor system.

A measure of the relative performance of a multicomputer system is the speedup factor, $S(M) = t_s / t_p$, where t_s is the execution time using one processor and t_p is the execution time using a computer with M processors. Yet, also, in a message-passing system, the time to send messages must be included in the total execution time of a problem. Thus, the parallel execution time (t_p) is obtained by adding two elements: the computation time (t_{comp}), and the communication time (t_{comm}): $t_p = t_{comp} + t_{comm}$.

As the COW is for resolving the optimization problem and isolated from any external network, the standard deviation of t_p is very small and can be ignored. Only mean times will be considered.

Figure 12 shows the average time (in seconds) for each sample time and the speedup obtained as the number of processors is increased. In accordance with the results obtained, the following general observations can be highlighted: (1) for coarse initial partitions in the space of states ($\Delta x = 0.5$), the speedup with less than five slave processors is very low. In Fig. 12(a) it can be observed how *Amdahl's limit* (maximum number of processors for solving a problem in the minimum time) is reached for $M = 4$. Therefore, the result with $M = 10$ is worse than the time obtained by a single processor. (2) For fine initial partitions in the space

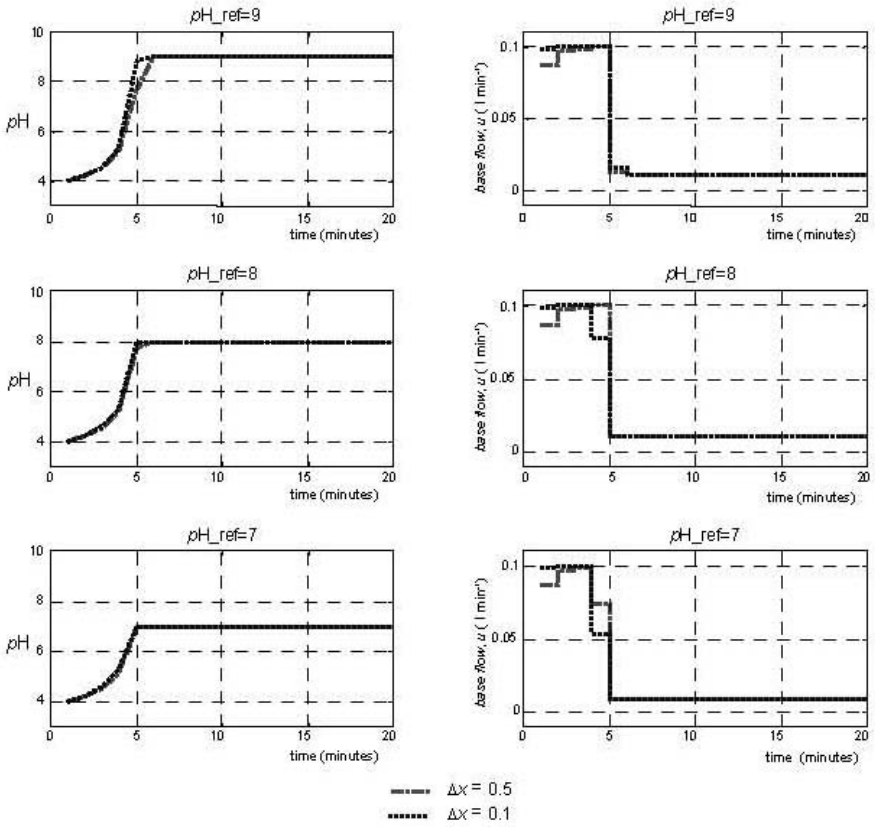

Fig. 11. Simulation results.

Table 4. Average time (in seconds) for each sample time in a single processor system

t (Δx = 0.5)	t (Δx = 0.1)	t (Δx = 0.01)
0.75	10	94

of states (Δx = 0.1), the computational load has been increased and the speedup reaches a saturation point from M = 13 as depicted in Fig. 12(b). (3) Finally, for very fine initial partitions in the space of states (Δx = 0.01), the speedup is quite better, almost linear (Fig. 12(c)). In this instance, since the computational load has been considerably increased, the computation part (t_{comp}) predominates over the communication part (t_{comm}) in the parallel execution time (t_p), as can be appreciated in Fig. 13.

One of the most important points to guarantee the viability of implementing the *p*H control in real time is the controller response time. It is clear that this time must be less than the sampling time. In this case study, the controller response time is 60 seconds [27] and the sampling time in a single

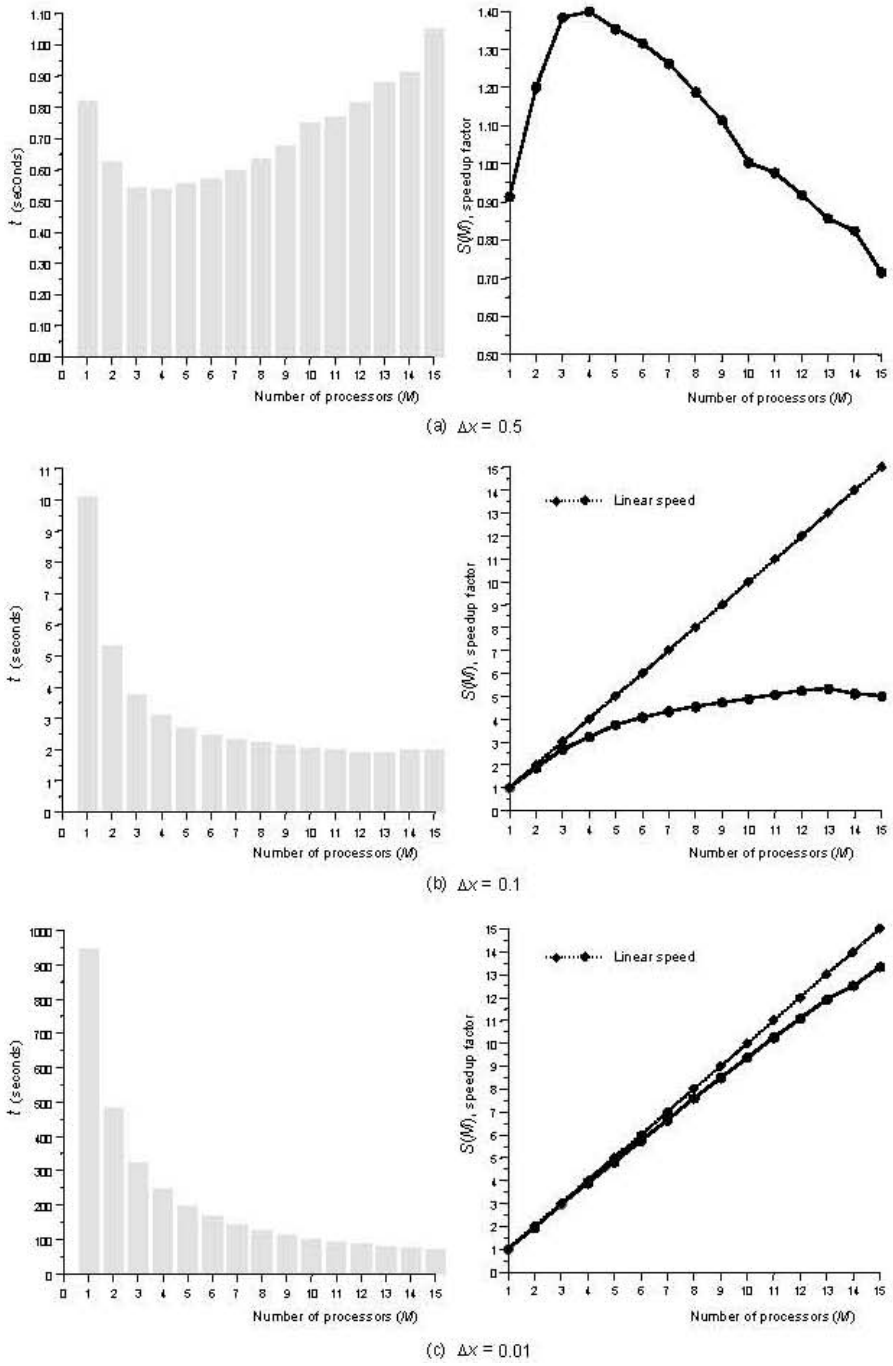

Fig. 12. Average times (in seconds) in each sample time and speedup factor.

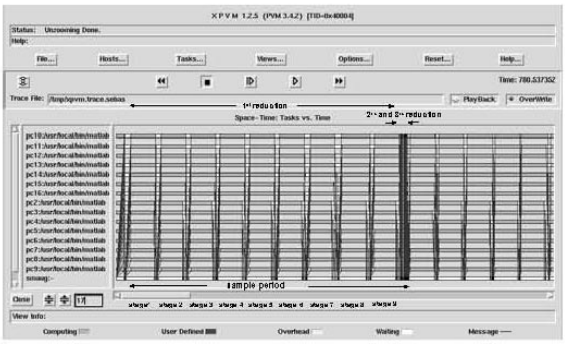

Fig. 13. Chronogram of *p*H control with an initial partition size $\Delta x = 0.01$.

processor is shown in Table 4. For $\Delta x = 0.5$ and $\Delta x = 0.1$ as sizes of the initial partition, the control of the *p*H could be solved in real time with just a single processor. However, with $\Delta x = 0.01$ it is necessary to use a cluster of PCs. In the experiment considered, with $M = 15$ the sampling time is 70 seconds (Fig. 12(c)), this means that by adding two or three processors to the cluster it is possible to control pH neutralization in real time.

CONCLUSIONS

This paper has shown the feasibility of using parallel processing methods to solve control algorithms in real time in the field of Control Engineering education. Decreasing prices and technological advances in personal computers have made it possible to carry out parallel processing in a simple and not-so-expensive fashion by building clusters of workstations (COWs). Now, low-cost parallel computers allow university departments to widen the range of processes to be controlled in the lab. Thus, new, challenging control projects can be included in traditional laboratory assignments.

In the experimental work presented here, it is demonstrated how *p*H process modeling and regulator design can be integrated into a cluster of PCs for a new control experiment in labs. It is further demonstrated that the experiments can be implemented in real time.

Currently, the lab is used just by a group of computer science students as a complementary activity. These students' comments, on the whole, have been very good since this practical experience gives them the opportunity of applying the theoretical concepts and algorithms that they have studied. The authors have observed that students are more responsive to algorithm implementation than theoretical details. Compared with students who do not use the control lab, those who do perform better in exams.

Acknowledgments—This work has been supported by the Spanish CICYT under grant DPI2007-61068.

REFERENCES

1. R. E. Bellman, *Dynamic Programming*, Princeton University Press, New Jersey, (1957).
2. R. E. Bellman and S. E. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, New Jersey, (1962).
3. A. Grama, A. Gupta, G. Karypis and V Kumar, *Introduction to Parallel Computing, Design and Analysis of Algorithms*, 2nd edn, Addison Wesley, (2003).
4. L. Cooper and M. W. Cooper, *Introduction to Dynamic Programming*, Pergamon Press, (1981).
5. A. J. Korsak and R. E. Larson, A dynamic programming successive approximations technique with convergence proofs, Part II, *Automatica*, **6**, 1970, pp. 253–260.
6. R. E. Larson, J. L. Casti, *Principles of Dynamic Programming, Part II: Advanced Theory and Applications*, Marcel Dekker, New York, (1982).
7. R. E. Larson and A. J. Korsak, A dynamic programming succesive approximations technique with convergence proofs, Part I, *Automatica*, **6**, 1970, pp. 245–252.
8. A. P. Madrid, S. Dormido and F. Morilla, Reduction of the dimensionality of dynamic programming: a case study, *American Control Conference* – ACC99, San Diego, USA, (1999).
9. A. P. Madrid, S. Dormido, F Morilla and L. Grau, Dynamic programming predictive control, *IFAC, 13th Triennial World Congress*, 2c-02, San Francisco, USA, (1996), pp. 279–284.
10. L. Moreno, L. Acosta and J. L. Sánchez, Design of algorithms for spatial–time reduction complexity of dynamic programming, *IEE Proc.-D*, **2**, 1992, pp. 172–180,
11. M. Sniedovich, *Dynamic Programming*, Marcel Dekker, New York, (1992).
12. R. E. Larson and E. Tse, Parallel processing algorithms for the optimal control of nonlinear dynamic systems, *IEEE Transactions on Computers*, C-22, **8**, 1973, pp. 777–786.
13. R. Buyya, *High Performance Cluster Computing*, Prentice Hall, vol. 1 (Architectures and Systems), New Jersey, (1999a).
14. R. Buyya, *High Performance Cluster Computing*, Prentice Hall, vol. 2 (Programmings and Applications), New Jersey, (1999b).
15. G. F. Pfister, *In Search of Clusters*, Prentice Hall, New Jersey, (1998).
16. M. Snir and W. Gropp, *MPI: The Complete Reference*, The MIT Press, Cambridge, Massachusetts, (2001).
17. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancher and V. Sunderam, *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*, The MIT Press, Cambridge, Massachusetts, (1994).
18. URL: http://www.uned.es, (2006).
19. F. C. Berry, An undergraduate parallel processing laboratory, *IEEE Trans. Educ.*, **38**, 1995, pp. 306–311.

20. T. Hintz, Introducing undergraduates to parallel processing, *IEEE Trans. Educ.*, **36**, 1993, pp. 210–213.
21. B. Wilkinson and M. Allen, A state-wide senior parallel programming course, *IEEE Trans. Educ.*, **42**, 1999, pp. 167–173.
22. J. A. Youssefi and K. Zemoudeh, A course in parallel processing, *IEEE Trans. Educ.*, **40**, 1997, pp. 36–40.
23. R. E. Larson, J. L. Casti, *Principles of Dynamic Programming. Part I: Basic Analytic and Computational Methods*, Marcel Dekker, New York, (1978).
24. S. Dormido-Canto, A. P. Madrid and S. Dormido, Parallel dynamic programming on clusters of workstations, *IEEE Trans. on Parallel and Distributed Systems*, **16**(9), 2005, pp. 785–798.
25. J. Fernández, A. Cañas, A. F. Díaz, J. González, J. Ortega and A. Prieto, *Performance of Message-Passing Matlab Toolboxes*, Vol. 2565, *I* Heidelberg, (2003) pp. 228–241.
26. J. M. Maciejowski, *Predictive Control with Constraints*, Prentice Hall, (2001).
27. S. D. Canto, Programación dinámica paralela: aplicación a problemas de control, Ph.D. thesis, Department of Computer Science and Automatic, (UNED), Madrid, (2002).

Sebastián Dormido Canto received his MS degree in Electronic Engineering in 1994 from the Universidad Pontificia de Comillas (ICAI), Spain, and his Ph.D. in physics from the Universidad Nacional de Educación a Distancia (UNED), Spain, in 2001. He joined the Department of Computer Science and Automatic Control of UNED in 1994 where he is currently an associate professor of Control Engineering. His research and teaching activities are related to: the analysis and design of control systems via intranet or internet, high performance interconnection networks for cluster of workstations and optimal control.

José Sánchez Moreno received his computer sciences degree from Madrid Polytechnic University in 1994 and his Ph.D. in sciences from UNED in 2001. Since 1993 he has been assistant professor in the Department of Computer Science and Automatic Control in UNED. His current research interests are: the design of new systems for control education, web-based laboratories, and distributed networked control systems.

Sebastián Dormido Bencomo holds a degree in physics from the Complutense University in Madrid, Spain (1968) and a Ph.D. from the University of the Basque Country, Spain (1971). In 1981 he was appointed professor of Control Engineering at the Universidad Nacional de Educación a Distancia (UNED), Spain. His scientific activities include: computer control of industrial processes, model-based predictive control, robust control, model and simulation of continuous processes and control education with special emphasis on remote, virtual labs and e-learning. He has authored and co-authored more than 200 technical papers in international journals and conferences. Since 2002 he has been President of the Spanish Association of Automatic Control, CEA-IFAC, where he promotes academic and industrial relations.